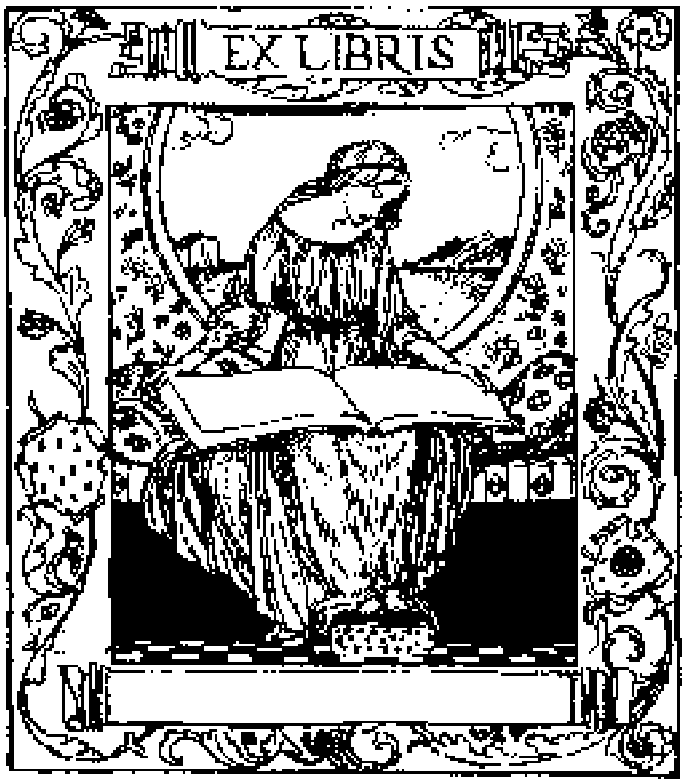
John's WordPerfect

Scripting Guide

John Rethorst



To my Students

Foreword

A search for a good word processor is well worth the time. When starting a Ph.D. program – for which I would be doing a ton of writing – I toured Word, FullWrite, WriteNow and MacWrite Pro, and then tried WordPerfect, at that time in version 1. It had been panned in reviews, but I wanted to see for myself. What I found was one of the friendliest and most competent programs I had ever used.

The reviews had taken the Mac version to task for being too DOS-like, and un-Mac like in other ways. "Too many hierarchical menus," went one criticism. But most programs of any complexity these days have more hierarchical menus than WP 1 did. "Mac users don't like codes" said someone else. Of course not, but Word now has them too – just because they can make some complex tasks much easier. I thought WP was a great program in many ways, but what I liked most were the macros.

When version 2 added a scripting language to the macro engine, I was hooked. Driven by a typical Mac user's quest for ultimate elegance, I found that I could add or fine-tune features with astonishing power and flexibility. Also, scripting was just plain fun.

Then I taught an introductory computer course and wrote a lab manual for it that turned into my first book, *Welcome to the Macintosh – From Mystery to Mastery.* Before I knew it, I had become hooked on teaching and writing about computers, as well as on the beasts themselves. Putting two and two together, I thought I would write a book on WordPerfect Mac. Version 3 was then in development, a massive piece of engineering representing WP Corporation's increasing commitment to the Mac marketplace. If I got busy, I could write about the program just in time for its release.

Although I knew the book would be fun, I didn't know I'd like doing it so much that instead of the 350 pages my publisher wanted, I gave them 550. The good people at Henry Holt were as supportive as ever, but we knew the market wouldn't want a book much larger than that. Also, as *Teach Yourself WordPerfect* was to be a beginning to intermediate guide, I was lucky to be able to include a few examples of WP macro and AppleScript code.

There was much more to write, and this is what turned up. A very special thanks for the collaboration of the World's Greatest Technical Editor, Dan Smellow, who also taught computers with me at Cornell, and is a better programmer besides. Thanks also to Scott Lawton, Allan Greenier and Gero Herrmann for their valuable advice on AppleScript.

This book assumes no prior knowledge of programming, but you should know how to use the Librarian to copy and install macros (see Appendix E otherwise). This tutorial then makes you an expert. Three chapters also discuss AppleScript with WP, and how AppleScript and WP macros interact.

I haven't duplicated the reference capabilities of the online help or the manual on the CD. This book contains lessons, with concepts, explanations and over 1700 lines of sample code, rather than an alphabetical list of commands and variables. The latter is a better ready reference for the seasoned user. Conversely, an attempt to learn how to write macros using a reference would be frustrating at best – just like learning a foreign language by a dictionary.

About this Book

Actual macro code is in Helvetica and indented.

Example syntax is in Helvetica and boxed.

Text examples are shadow-boxed.

Many of the examples here are excerpts from the macro sets listed in Appendix D. If you're working with a chapter that discusses one of these sets, I recommend downloading that set to use as a more complete reference than space allows here.

Writing, line drawing and book formatting done in WordPerfect, and printed to a reader by eDoc.

Comments welcome, at jcr2@cornell.edu.

Table of Contents

Illustrations

1: Starting Out

Welcome to WP macros! This feature is one of WordPerfect's most powerful, and it's also one that takes a little while to learn. Macros aren't hard, though, and with a little practice you can *significantly increase the power and flexibility of your work environment.*

First of all, what's a macro? It can be thought of in two ways: as a tape recording, and as a list of instructions. The easiest way to use macros is like a tape recorder: start a recording and, instead of talking or singing, do some things in WordPerfect: open a new document, say, and set font and margins. Then stop the tape recorder. Now, any time you play that recording, WordPerfect will repeat those exact steps – but very quickly. Another advantage becomes apparent with longer sets of steps: since we humans tend to get bored by repetitive work, accuracy can suffer. With a macro, if it's right the first time, it's right every time.

While the program's "recording," it's actually writing a list of what you're doing. A more sophisticated use of macros involves editing such a list, also called a **script**, or even writing one from scratch. A *big* advantage to that is being able to tell WordPerfect to do one thing *if* a certain condition exists, and *else* do another thing. This way you can actually have the program do some of your thinking for you, rather than just sequences of actions. When you finish this book, you'll be able to write powerful and elegant commands that make your word processing easier, more productive, and more fun!

You'll also be able to use Apple's system-level scripting language, called *AppleScript*, to write instructions that make WordPerfect and other programs work together, or expand WP's capabilities still further. You'll be astonished by how much you can do – and by how easy it is.

In this first chapter, we'll record a simple macro just to work with, and then learn how to edit it. Future chapters will get very advanced, as we work our way through WordPerfect's macro environment. You'll be surprised how easy it is.

Recording a macro

As setup, open an existing file and select a few lines of text.

1. From the **Tools** menu, choose the **Macro** submenu and then the first item, the **Record** command. A dialog box like figure 1 appears. (Press the Command key to see the keyboard equivalents shown.)

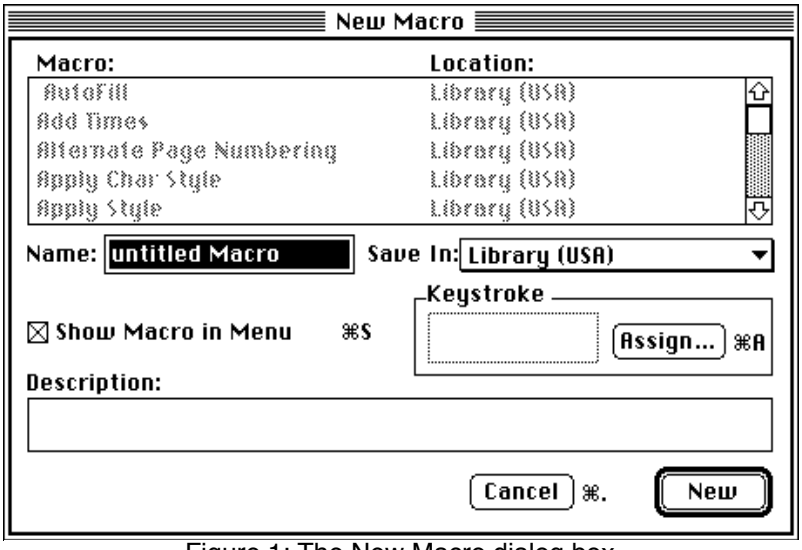


Figure 1: The New Macro dialog box

This dialog has the name "untitled Macro" selected. Type in "Copy to new file" and click **New**.

The dialog box disappears, and WordPerfect is now recording. It does not record the time between each step, only the steps themselves. So there's no hurry.

2. Choose **Copy** from the Edit menu.
3. Choose **New** from the File menu.
4. Choose **Paste** from the Edit menu.
5. Choose **Select All** from the Edit menu.
6. Choose **Times** from the Font menu.
7. Choose **24** point for a size.
8. Press the **right arrow** key.
9. From the Macro submenu of the Tools menu, choose the **Stop Recording** command (it's now the first item).
10. Click the **Save** button in the resulting dialog box, to save this macro.

Now, close your new document (don't bother to save it), and select some other text in your original file. Then go to the Macro submenu of the Tools menu, scroll down the alphabetical list to your new macro, and run it.

That's fun, but also a timesaver and a real help for accuracy. It's also only a hint of what you can do. It's as far as a lot of people want to go: record and play back. Let's take the next step, though, and *edit* the commands that WordPerfect recorded.

Editing a macro

1. From the **Macro** submenu, choose the third command, **Edit**. A dialog box like figure 2 appears:

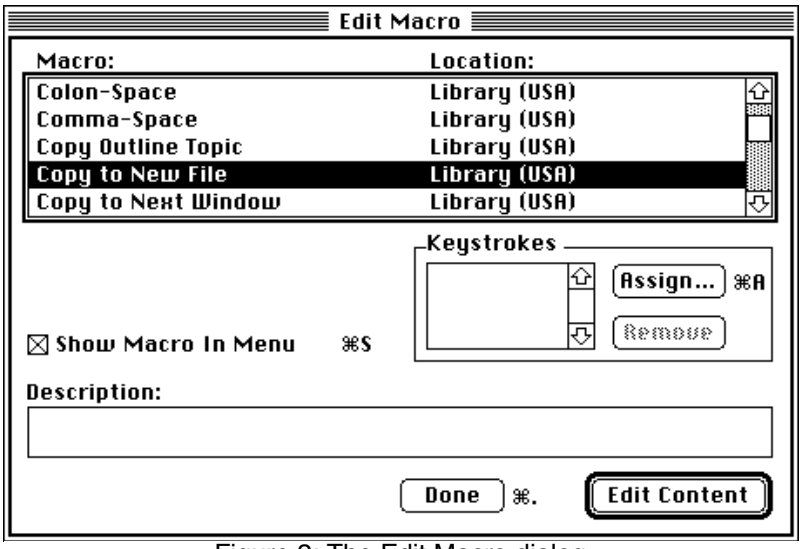


Figure 2: The Edit Macro dialog

2. Scroll to your first macro, "Copy to New File," and click on it to select it (you can also type the first few letters of the name to jump to it).

3. Click the **Edit Content** button, or press **Return**. You can also double-click on the name. A new window opens, looking like figure 3:

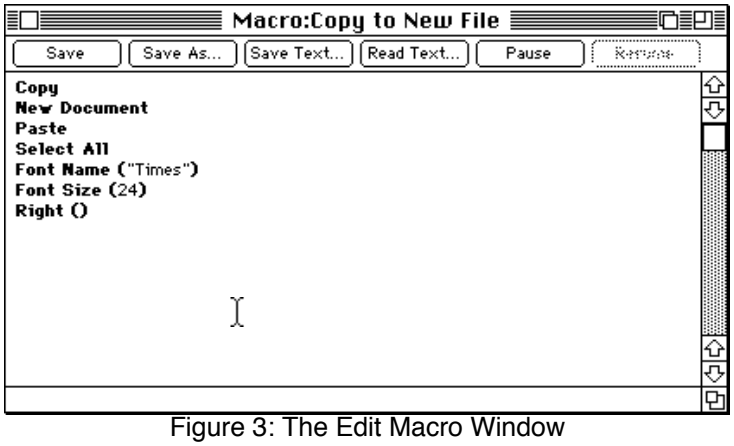


Figure 3: The Edit Macro Window

4. and which looks a little complicated at first. As you see, though, it's just the list of commands you did a moment ago.

Note that the fifth line reads: **Font Name (**"Times"**).** Double-click on the word "Times" so that only that word (not the surrounding quotes) is selected.

5. Type **Helvetica** so that the word fits in the surrounding quotes.

Note that the sixth line reads: **Font Size (**24**)**

6. Double-click on the number **24** so that only that number is selected.

7. Type **72** so that the number fits in the existing parentheses.

8. Click the small **Save** button near the top left of the macro editing window.

If you did not do these steps correctly, an error message will appear. OK that, then click the close box at the top left of the macro window, don't save changes, and start again at step one of this section. If you did these steps correctly, no message will appear, but the Save button will gray out. Click the close box at the top left of the macro editing window, and you're set. Select some text in the document, and run your edited macro.

As info, the words "Font Name" form a command, and the word "Times" is the **parameter** or the **argument** for that command.

Congratulations on what you've learned. In the next chapter we'll learn how to enter commands from scratch in the macro editing window, to do things it's just not possible to record. We'll work on a more advanced topic each chapter, until the 900 macro commands and variables within WordPerfect give you a new dimension to the phrase "Power User."

2: Scripting a Macro

Now that you've edited a macro, you're more advanced than most WordPerfect users. You have substantial power at your fingertips already, and one next step might be to watch yourself work with the program for a week, and take note of how many sets of steps could be more easily done with a macro.

You may find that there are several tasks that could profitably be automated, and some of them don't seem accessible to recording. One of the most important is having a macro make a decision, based on what's in your document.

For example, the Copy to New File macro we did last chapter works on text you already have selected. Try running the macro without selecting any text first. Oops. But we can script a macro that will look on its own to see if any text is selected. If text is selected, the macro will do one thing; if not it will do something else. These are important words, *if* and *else*. They're exactly the words we'll use.

We'll also use a specific way to ask WordPerfect to look for selected text. The program's macro language gives us this way in the form of a **flag**, so to speak. WordPerfect puts up a flag if text is selected, no flag if it isn't.

So we can write the macro to look for that selection flag and, if it's there, do one thing, or else, do another thing. For starters, let's just have the macro tell us whether any text is in fact selected. We can have WordPerfect give us an alert saying that one condition or the other is true. While we're at it, we can have the program beep at us, so we'll be sure to notice the alert.

The script, or text, of a macro

Let's look at the exact wording, or the **syntax**, of our script. Here it is:

```
If (SelectionFlag)
    Beep
    Alert ("Yes, some text is selected.")
Else
    Beep
    Alert ("No, no text is selected.")
End If
```

and while it's not exactly good English, it's legible and, after a little exposure, logical. You see that it starts with an If statement: what to do in the event it finds the selection flag to be on.

The next two lines are instructions to be followed should the If condition be true. Beep, and then display what WordPerfect's macro language terms an **alert**, and which will look like figure 4:

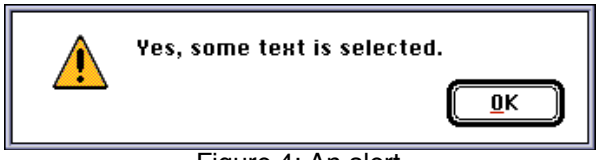


Figure 4: An alert

and that's the end of what the macro will do in case the If condition is met. The next section of the macro, Else, describes what the macro will do otherwise. These are the only two choices, because text is either selected or it isn't – no middle ground.

The Else statements are similar to the If statements in our example, but could be entirely different, according to our needs.

The macro ends with an End If statement, to let WordPerfect know that the If/Else situation is over.

Using the macro editor

To enter this macro:

1. Choose **Record** from the **Macro** submenu of the **Tools** menu, just as though you were going to have WordPerfect watch and record your steps. Give your macro a name, maybe "Selected Text?" and click **New**.

2. From the **Window** menu , choose the macro window. It will be at the end of the list.

3. You now have a macro script window in front, and it's blank. This window is called the **macro editor**.

4. Without tabbing or otherwise indenting, type each line as you see it above, pressing Return at the end of each line except the last one. WordPerfect will indent and bold the text properly for each line, as soon as you move to the next line.

If anything you type appears underlined, after you've pressed Return for that line, you've made a typing mistake. Check carefully for misspellings, lack of spaces or missing parentheses or quote marks.

5. When you finish typing, click the **Save** button at the top left of the window. The last line of the macro, "End If, " should change to bold as have the other commands. Your macro script should look like figure 5:

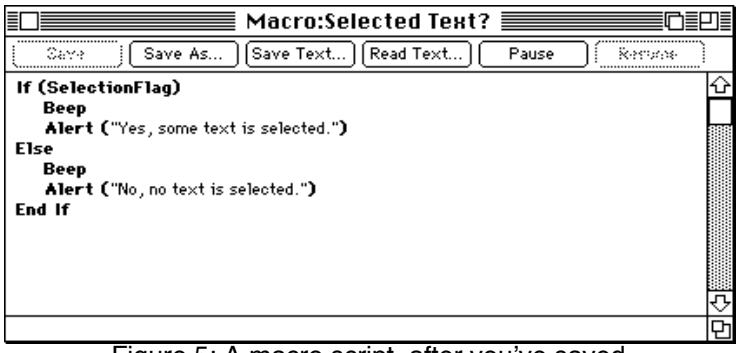


Figure 5: A macro script, after you've saved

If you haven't corrected any underlined text, though, a window like figure 6 appears:

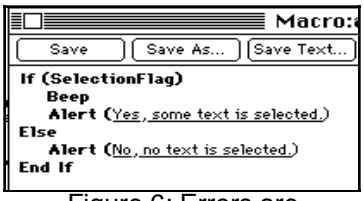


Figure 6: Errors are underlined

showing that, in this example, I typed the parameters for the Alert command in parentheses, but not in quotes. (There's good reason for all this punctuation, as you'll see.) If you don't correct all the underlined text, and instead just click the Save button, a dialog like figure 7 appears:

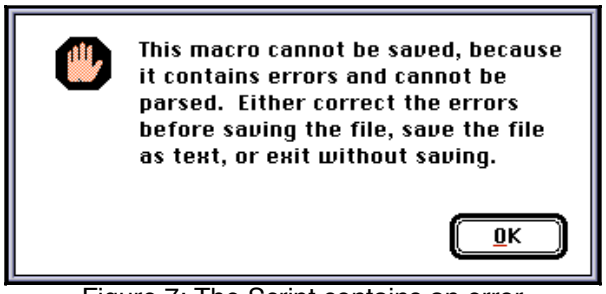


Figure 7: The Script contains an error

which you can OK and then check your script again. After you fix everything and save it:

6. Close the macro editor by clicking in its close box. Back in your document, select a few lines of text, and run the macro. Then try it with no text selected.

What it all means

This is just an initial glimpse of how a macro can interact with what you have on screen. You can have macros do one thing if your text is in one font or another, bold or italic, if your document is longer than a certain number of pages, or any number of other possibilities.

Another way of looking at this is that you're having the macro do some of the thinking for you. In fact, it's doing the low-level thinking, freeing you for the more interesting parts of your work.

Congratulations again! You've done what is in truth some fairly sophisticated programming. Your macro **branched** according to a **conditional statement**, pretty high-powered stuff, and clearly more than could be recorded.

In the next chapter we'll look at ways for a macro to make a more sophisticated decision according to what's on your screen. We'll get practical as soon as we possibly can, and you'll be developing *tools you can use*.

3: Scripting with Variables

In the last chapter we looked at how to use a flag in a macro, and how to have a macro make a decision for us, based on conditions in the document. This chapter we'll look at something similar to a flag, but more versatile, called a **read-only variable.** This gives us a little more information about the document, which we can use the same way in an If/Else statement. We can also put the contents of the variable itself in an Alert.

Read-only variables

In the courses I've taught, the main thing students seem to want out of their papers is that they be long enough. Five pages equals significant thought, four pages doesn't. A student could use a macro to test whether a paper is at least five pages long:

```
End ()
If (PhysicalPage>4)
    Alert ("Yay! You're finished.")
Else
    Alert ("You gotta keep going.")
End If
```

which would all be typed into the macro editor – nothing except the first line could be recorded. To enter this script: start a macro recording, with a title something like "Enough Pages?" Then:

1. From the Window menu, switch into the macro editor.

2. Type the script as shown above. After you press Return for each line, WordPerfect will bold some words and indent some lines. If it underlines any words, you've made a typing mistake.

When done, your macro editor should look like figure 8:

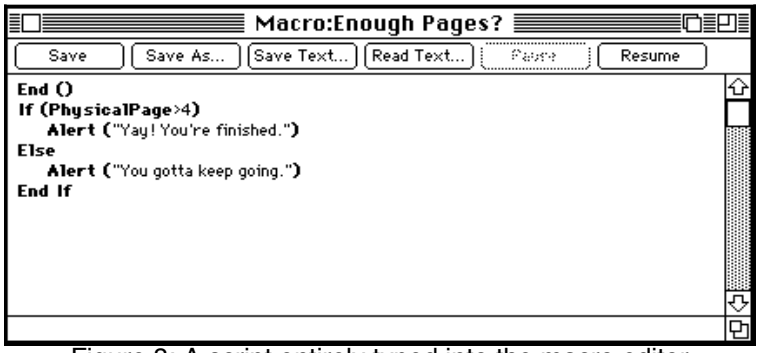


Figure 8: A script entirely typed into the macro editor

3. Close the macro editor, saving changes. Try this macro on documents that are either more or less than five pages long.

Let's look at the grammar and vocabulary (or the syntax) again. The first line instructs WordPerfect to go to the end of the file, since it's the length at the end that we want to measure. The empty parentheses could contain the word "select," in which case the macro would select all the text as it goes to the end. As it's written here, the insertion point will go to the end of the file without selecting anything. It's just vocabulary.

The second line has the If statement, and the word **PhysicalPage**, which means the page of the file the insertion point is on. Vocabulary again, and much like SelectionFlag, but let's notice a difference.

The term SelectionFlag can be called a flag since it's either up or down; the condition is either true or false. PhysicalPage is not a flag, since it's not a matter of yes or no; rather, one of quantity. What both have in common is that they vary according to what's going on with your file, so we call them **variables**.

In both this example and in the one we used last chapter, the If statement tests the value of a variable. The test in this case uses the standard "greater than" symbol, or ">" and could just as well have used the less than "<" symbol or the equals sign.

Or could it? If the test were "If (PhysicalPage=5)" and our hapless student had already written six pages, what would the macro have told him or her? Right – to keep going! This is a big point to consider with macros: if the syntax is right, the script does *just what you tell it to*. Often we overlook an error in logic because we, as humans, are used to making assumptions, often false ones, to our detriment. So another advantage to learning macros is that you'll win more arguments.

More sophistication

While this macro is helpful to someone who needs to write at least five pages, it could be more helpful by telling him or her how many more pages must be written. Let's have the macro calculate that for us. Edit your script so that it looks like figure 9 (again, WordPerfect will do the bolding and indenting for you, and underline any mistakes). You can use copy and paste for the If, Alert and End If lines, and change just a couple of numerals each time.

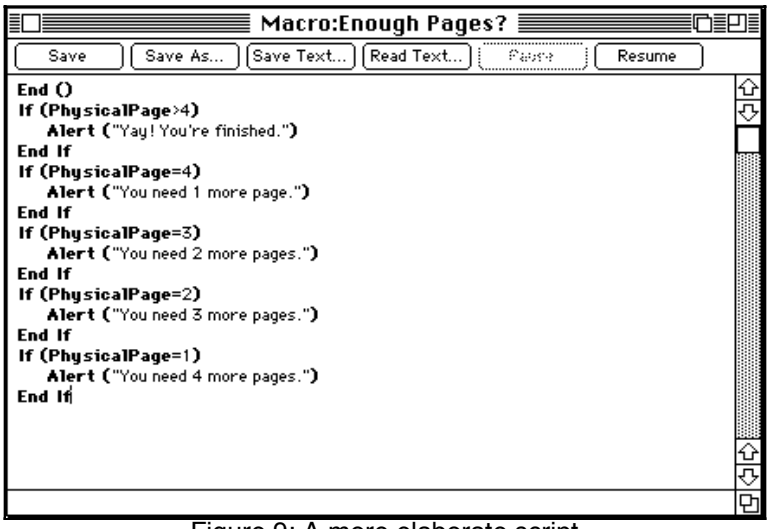


Figure 9: A more elaborate script

Run this macro on a document or two. Note how the If statements work: if the first condition is true, the macro displays the first alert, and so on. Another way to design this macro would tell the user not how many pages are needed, but how many pages are already there. We can do that by adding the variable to the Alert line, as in figure 10, which uses something new to us in macros, the dollar sign. We call this an **operator** and, in scripts, it means **join.** The first and last parts of the second alert are thus regular text, in quotation marks, joined to the read-only variable.

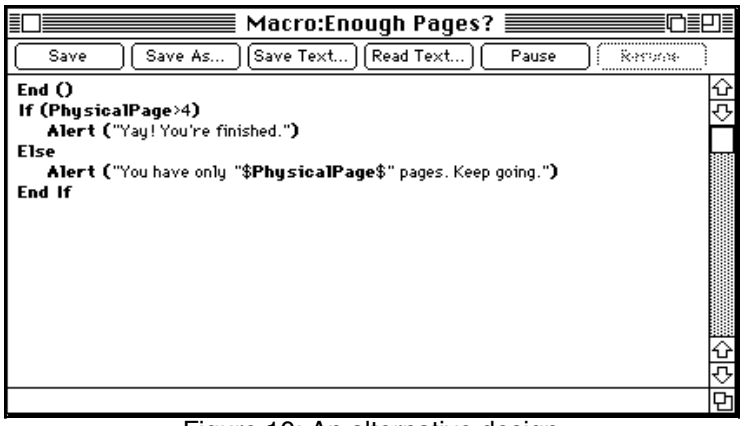


Figure 10: An alternative design

Note that in the regular text, the word “only” has a space after it, before the quote marks, and the word “pages” has a space before it, after the quotes. This just puts spaces on either side of the page number.

This kind of design, providing the user more information, can be very helpful when the information is otherwise less readily available. Using read-only variables and flags lets you script some very complex decision patterns, and provide the user a lot of data as well.

Congratulations once more! You're learning a lot. In the next chapter we'll look at another kind of variable, one that you can put data *into* as well as get data from. It's hard to describe at this point the power and flexibility you'll have with these tools.

4: Read-Write Variables

In the last two chapters, we looked at variables that the program sets, according to whether you've selected text or written a certain number of pages, and all the macro can do is read the value of those variables. So, we call these read-only variables. We use these a lot in macros, but WordPerfect's power is substantially enhanced by another kind of variable, one which a macro can write the value of as well as read it. We call these **read-write variables**. Here's an example.

Putting a number in a variable

Let's expand on the flexibility of the "Enough Pages?" macro that we wrote last chapter. It was fine for an assignment of five pages, but you know those professor – they change their assignments all the time. So we need a macro that will first ask the user how many pages are required, then take that value and use it for the If/Else test.

Since we're modifying our "Enough Pages?" macro, this is also a good place to try the **Save As** button in the macro window. So:

1. From the Macro menu, choose Edit, then choose "Enough Pages?"

2. Click the Save As button, and give this new macro a different name.

3. Edit the script so that it reads just like this, but note that the first line is a long one: don't press Return until just after you type "have?)"; it's OK if the text wraps to another line as you type:

```
Get Integer (Var01;1;100;"Number of pages";"How many pages do you need to have?")
End ()
If (PhysicalPage>Var01)
    Alert ("Yay! You're finished.")
Else
    Alert ("You gotta keep going.")
End If
```

4. Click the Save button, and close the macro editor.

Let's see what the syntax means. The first line starts with the command **Get Integer**, which has the macro ask the user for an integer. Within the parentheses, the first parameter, **Var01**, is just that: variable no. 1 (read-write now, so it has no proper name – it will contain what we assign to it, in this case whatever number the user types).

The next numbers, 1 and 100 separated by semicolons, are a check WordPerfect makes on the range of the integer the user enters.

The next part of the entry has to be in quotes, and is the title of the dialog box that will ask for the integer. The last entry, also in quotes, is the text of the dialog box. When you run the macro, that will look like figure 11:

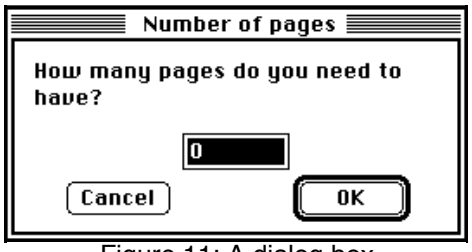


Figure 11: A dialog box

The only other difference in this modified macro is the If statement, which tests the physical page against Var01, which now contains the required number of pages.

5. Run this macro a few times on some appropriate test documents, and enjoy your new computer abilities. Test it by entering a number outside the 1 to 100 range.

A little mistake

While we're generally doing very well, we let a small error creep in a minute ago. Did you see it? Hint: it's not as bad as the error we discussed last chapter, which told the student with six pages to keep going to reach five.

In this case, the macro test was accurate when it tested for five pages by saying If (PhysicalPage> 4), but when we put in Var01 and then tested for a value greater than that, we made the student write one more page than necessary. To correct that, change the third line to:

If (PhysicalPage>=Var01)

so that the operator means either greater than or equal to. The standard symbol puts the 'greater than' part above the equals sign, but the Geneva font isn't that fancy, so WordPerfect has us type the operator as shown.

Another example

Let's script a macro that you might find helpful as long as you use WordPerfect. As you know, you can set measurements in the program in inches, points, or several other units. Clicking on any unit of measurement in a dialog box will produce a menu of the others. Default is set in Preferences.

I leave the default at inches, since that's the unit I normally think in for margins, indents and so on. But for leading, the distance between two lines of text, I'd rather use points, since type sizes themselves are measured in points. We can open the line spacing dialog box and, every time, open the hidden menu to change units — or how about a macro to set leading in points, while we leave inches as the default for everything else?

The tools we'll use are: one read-only variable: **CurrentLeading**, one read-write variable: **Var01**, and three commands, **Get Integer**, **Leading** and **Automatic Leading**. I'll have more to say soon about how we *find* these commands and variables (hint: they're all in the online macro help).

A little conceptual mapping first. What we want to do is:

1. Get the leading, measured in points, that the user wants.

2. Put that value into a variable.

3. Tell the program to set the leading to that variable.

Remember that the Get Integer command has places for lower and upper limits as a check. WordPerfect's limits on leading in points are 1 to 32767, so let's use those.

Start a macro recording, name it something like "Leading in Points" and, from the Window menu, switch to the macro editor. Type this first line. It's all one line, even if wrapped, so don't press return until after typing "to:")":

```
Get Integer (Var01;1;32767;"Set Leading in Points";"Set leading to:")
```

which would, just as in the Number of Pages example above, ask the user for a value, and put that value into Var01. We'd then set the leading to that value with the line:

```
Leading (Var01)
```

and those two lines will do it! Save this script, and check out your new macro.

More elegance

Since we're getting to be real programmers now, let's look at a couple of helpful additions. First, let's tell the user what the current leading is, by adding the read-only variable CurrentLeading to the first line. Note the join operator ($) that we learned last chapter. Also, remember to put spaces before the quote mark preceding the variable and after the quote mark following the variable, so the dialog box will have spaces before and after the value:

```
Get Integer (Var01;1;32767;"Set Leading in Points";"Current leading is "
$CurrentLeading$" point(s). Set to:")
```

which tells the user the current leading. Note: this is all one line in the macro editor. Save and close the script, and try the macro now (testing a macro as you write it isn't a bad idea. When building a complex script, this can save some time).

For a final touch, let's remember that WordPerfect defaults to automatic leading. The value of that depends on font and size so, when our text is set to automatic leading, the CurrentLeading variable would say 0 points for 9 pt. Geneva, or 3 pts. for 36 pt. Geneva, and so on. Let's use an If/Else statement to let the user choose either automatic or another value. We could let a value of 0 equal automatic leading, and our macro would then look like:

```
Get Integer (Var01;0;32767;"Set Leading in Points";"Leading is now "
$CurrentLeading$" point(s). Enter 0 for automatic leading, or set to:")
```

```
If (Var01=0)
    Automatic Leading
Else
    Leading (Var01)
End If
```

and, since the Get Integer dialog defaults to 0, the user would only have to run our macro and click OK to get automatic leading.

And there you go. You're off to a great start to adding commands that you want to use, so your word processing, graphics and page layout will be faster, easier and more accurate.

5: Flow Control

Last chapter we learned how to use a read-write variable to take a number the user gives us, and issue a program command with that variable, or compare the value of the variable to another variable. In one example last chapter, we compared the value of Var01 (read-write variable no. 1) with the value of a read-only variable, PhysicalPage.

This time let's compare the value of two read-write variables. You'll enjoy the structure of this, and it's a good first glimpse into the power you get by having variables work together. We'll also learn how to write a **repeat loop** – macro commands that repeat, or loop back to the beginning and run again, a certain number of times.

Writing a repeat loop

For a simple example of a repeat loop, try this. Note that there's a space after the period in the second line:

```
Repeat
    Type (I will not write macros in class. )
Until (PhysicalPage>1)
```

When you enter this in the macro editor, the line that repeats will indent automatically, and commands and variables will be bolded.

Try this in a new, empty document. Note that the test in the third line is much the same as the test in the "Enough Pages?" macro.

Planning the script

Now, for a more useful command to script, consider that WordPerfect lets you write some words, and then count them. What if, though, you wanted to go the other way and select a given number of words?

Before starting to code, let's put together a rough plan of what we need to do. Basically, that is:

1. Get a number from the user.

2. Start selecting words, counting as we go.

3. Stop when we've selected the number of words the user wants.

We'll do the first step with a Get Integer command, just like both examples last chapter. We'll set the limits of the Get Integer command to check from 1 to 32,767 (which happens to be the largest number a variable can hold).

To select words, we'll use a navigational command, specifying selection. This is similar to the "Enough Pages?" macro, where we went to the end of the document with the command End () which took us to the end without selecting anything. If the command were **End (Select)**, every-

thing from the insertion point to the end of the file would be selected. The navigational command for what we want to do this time is **Word Right (Select)**.

Now the fun part. We'll put the number of words the user wants to select in Var01. We have the Word Right command, which we'll put in a repeat loop. How do we tell the macro when to stop? By adding a line to the repeat loop that counts the number of times it's repeated. Then we test whether it's repeated the number of times the user wants. Here's the script:

```
Get Integer (Var01;1;32767;"Select Words";"Enter number of words to select:")
Repeat
    Word Right (Select)
    Assign (Var02;Var02+1)
Until (Var01=Var02)
```

which gets the integer, starts the repeat loop, selects successive words starting at the insertion point and going right and, using the **Assign** command, increments Var02 by 1 each time around, so that Var02 will contain the number of words already selected. When that number equals the number the user wanted, the macro stops. Voilà.

Adding some polish

We can make this go much faster, a real benefit when selecting a large number of words, by beginning the macro with the line:

```
Display (Off)
```

which turns off display while the macro runs, so it can select and count words without taking the extra time to update the screen while doing so. End the macro with:

```
Display (On)
```

Try this. There's quite a difference.

Variations

This is fine to select a quantity of words. What, you say that what you really need is to select a given number of *lines?* No problem. Just open the script, Save As under another name, say "Select Lines," and replace Word Right (Select) with **Down (Select)**. Quick and easy.

What happens if what you wanted to select were *groups* of lines, such as addresses? You might have a list of 200 names and addresses, and want to select the first 60 – or the 60 immediately following the insertion point. Let's talk strategy here. We have to figure out how to tell the macro what counts as an address. Words and lines were easy, since WordPerfect knows what those are. We couldn't just write something like "Address Down (Select)," though. How can we have the macro *detect* an address?

Consider that nearly all lists of addresses have a name on the first line, street and number on the second, and city/state/zip on the third, and maybe phone number on the fourth – or something like that, with a single hard return for each line. Then, to separate one entry from the next, there are two hard returns. That's our key, right there. Two hard returns.

Rather than figure out how to look for two hard returns in succession, let's have WP do some of the work for us. You know that you'd search for double hard returns in a regular document using Find/Change. The last menu in that dialog box, Insert, lets you find tabs, hard returns, etc. or any combination. Let's record that part of our macro, to replace Down (Select).

Recording part of a script

This is much like what we did in chapter one, only in reverse. Then, we recorded a change in font and size, and edited those values. This time, we'll put the insertion point at the appropriate place in the script, switch out of the macro editor, call the Find command, set its menus, and click Find. WordPerfect will record this as we go, and we can then go back into the macro editor and clean up a little. Ready? Follow these steps:

1. In the macro editor, click to put your insertion point at the start of the line "Down (Select)."

2. Click in the document behind the macro editor, to bring the document to the front, or do the equivalent using the Window menu.

3. Call the Find/Change command from the Edit menu. Set the Direction menu to Forward, the Where menu to Document Only, the Action menu to Extend Selection and, from the Insert menu, choose Hard Return twice, to put those codes in the Find box. Click Find. It doesn't matter if the command finds anything at this point. Close the Find dialog.

4. Switch back into the macro editor. You should see these lines, in the middle of the script you wrote:

   ```
   Find/Change Direction (Forward;No Wrap)
   Find/Change Where ({Current Doc})
   Find/Change Match (Partial Word;Case Insensitive;Alphabet Insensitive;CharRep
   Insensitive;{Text Only})
   Find/Change Action (Extend Selection)
   Find String ("[Language:English (USA)][Font:Geneva][Size:12][Hard Return][Hard
   Return]")
   Find
   Abort When Not Found
   ```

   and we need to edit this a little.

5. Change the line that starts with Find String so that it reads:

   ```
   Find String ("[Hard Return][Hard Return]")
   ```

   so as to delete the specifications of language, font and size.

6. The line "Down (Select)" will still be in the script. Delete that as well, and delete any blank lines that the recording may have added. The macro editor likes to put semicolons at the start of blank lines, for reasons we'll get to. For now, go ahead and delete those semicolons as well.

7. Save As with a new name, say "Select Groups" and check out your latest macro.

And that's it! You now have macros that select a given number of words, lines, and groups of lines. This last one would be useful for paragraphs as well as addresses, if you put double hard returns between paragraphs. More importantly, you're learning how to structure things. Repeat loops and other, similar **flow commands** are used all over the place in macros. As well, you know when to script and when to record. We'll be moving faster in future chapters, now that you have this grasp of the basics.

6: Text Formatting

Doing a large amount of identical formatting is a great job for a macro. Consider a glossary or dictionary (or catalog or handbook) where you want each word or phrase being defined in bold, and the description in plain text, like this:

> **geek**: person whose life has been taken over by computers, and who can't think about anything else.
>
> **hacker:** person who's very productive with computers, and who can use them to great advantage.
>
> **twelve-step program:** means to achieve freedom from something addictive and debilitating, such as computers.

One way to do this would be to select bold every time you started a new definition, type the word or phrase, and then switch back to plain text for the description. Dullsville and slow. Another way would be to use a table, put words in one column and definitions in the other, and bold the left column with one command. That would leave a lot of white space if the definitions were long, though. You could then do table to text, after replacing all colons with tabs, and then replace tabs with colons again, but this is starting to get complicated. Can we just enter the whole thing in plain text, and have a macro bold every word or phrase being defined? You bet.

We need to pursue a little strategy to tell the macro what to do. First, let's say that we used double hard returns to separate paragraphs (I know that paragraph spacing is a better way, but fewer people use that. It would be just as easy to design a macro for it, though). For double hard returns, we'll tell the macro to search forward until it finds the next pair of hard returns. We then want to select everything between those hard returns and the next colon, and change that selection to bold.

Recording the first part

To start off, begin a macro recording, and open the Find dialog. Set the options to Direction/Forward – No Wrap, Where/Document Only; Match/Text Only; Affect/Case and Text Only; Action/PosItion After. From the Insert Menu, choose Hard Return twice. Click Find. It doesn't matter if you find anything at this point. Switch to the macro editor; you should have this script:

```
Find/Change Direction (Forward;No Wrap)
Find/Change Where ({Current Doc})
Find/Change Match (Partial Word;Case Insensitive;Alphabet Insensitive;CharRep
Insensitive;{Text Only})
Find/Change Action (Position After)
Find String ("[Language:English (USA)][Font:Geneva][Size:12][Hard Return][Hard
Return]")
Find
Abort When Not Found
```

which gets us to the double hard returns, positioning the cursor after them.

First, let's edit the line that starts with "Find String" so it reads:

```
Find String ("[Hard Return][Hard Return]")
```

Now, add these lines to the script:

```
Find/Change Action (Extend Selection)
Find String (":")
Find
Abort When Not Found
Attribute (On;Bold)
Right ()
```

to extend the selection to the next colon, use the **Attribute** command to turn bold on for the selection, and move the cursor one character to the right without selection, to deselect the text that's been bolded. Try this on some sample text like definitions above.

This works, but it's rough. It ends with an error message when the macro can't find either double hard returns or a colon. To get it to repeat, you'd need to **Set Repeat Count** (Command-Shift-Clear) to at least the number of times you wanted it to repeat, then call **Repeat Count** (Command-Clear), then run the macro. Let's do better, by putting all this code in a repeat loop, and deleting unnecessary lines.

First, delete both lines that read "Abort when not found" and the last line, "Right ()." That last line deselects the bolded text, a nice feature for a macro that runs once, since if a macro or other action leaves anything selected, the user could inadvertently press another key and erase the selection. We don't want this line in a repeating macro, though, since each line of code takes a certain amount of time to run.

Then, add the line

```
Repeat
```

as the first line, and the line

```
Until (FindStatusFlag=0)
```

as the last line. **FindStatusFlag** is a flag just like SelectionFlag, which we learned in chapter two. FindStatusFlag is on (or equals 1) if the most recent Find command found something. So this macro runs until it runs out of things to find. Give this a try on your sample file. You would want to run it at the start of the definitions.

Oops – Bugs

Or would you? If the macro didn't start soon enough in the file to find double hard returns before the first word being defined, you'd miss bolding the word in that first paragraph. No sweat, just put the cursor at the top of the file, preceding the paragraph or two of introduction (for example, at the start of this chapter) and run the macro from there. Oops. That's much worse. What happened?

The macro did just what you told it to do. It found the first set of double hard returns (which, in this chapter, separate the two introductory paragraphs). It then found the first colon, which comes after "geek," and selected everything in between. Just what we *thought* we wanted.

At this point you, like all other Mac programmers, could say that you have something that works well enough, as long as you keep its quirks in mind. This is called a "hack." Or, you can fix it and call it "elegant." Up to you. Being WP users, let's go for elegance.

Let's see. We need to tell the macro to find double hard returns, signifying the start of a new paragraph, and then to look at that paragraph, to see whether it has a colon in it. If so, we can assume it's a definition. If not, we'll have the macro move on to the next paragraph. It's true that there may be a colon in an introductory paragraph, but unless we use a special symbol between word and definition, we don't have anything else to tell the macro to look for. We'll have the macro end when it can't find any more double hard returns.

This is best represented by a flowchart like figure 12. Flowcharting is a great way to conceptualize, design and troubleshoot macros. As we go along, I'll emphasize this way of thinking about the programming process, and you'll find it helpful to flowchart your way through variations on what you see here, and on your own ideas.

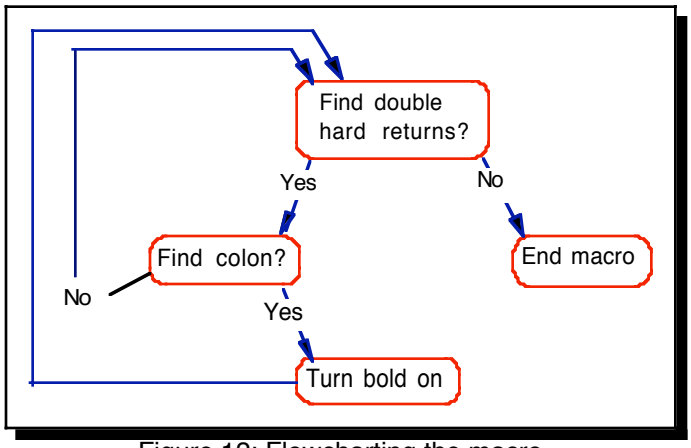


Figure 12: Flowcharting the macro

I'll also start adding **comment lines** to scripts. These lines start with a semicolon or have a semicolon after the command. Anything following the semicolon is ignored by WordPerfect when running the macro. They just serve to clarify the code to another user. Thus,

Assign (Var02;Var02+1) ; this increments the value of Var02 by 1

has a descriptive comment after the command. You can also start a line with a semicolon, followed by equals signs, hyphens or whatever, as a separator – again, to make it easier to read.

Now to the script. It makes two decisions now, both based on what it can find. So we're using FindStatusFlag twice, and thus don't want a repeat loop to continue until FindStatusFlag=0. It may well equal 0 at several points in the macro, but we want the macro to keep going. So we'll

add a **Go** command, which sends macro execution to a **Label**. Big words? Here's a quasi-macro example:

```
If (Today=Saturday or Sunday)
     Go (Party)
Else
     Go (Work)
End If
;============================
Label (Party)
Dance a lot
Make merry
End Macro
;============================
Label (Work)
Work hard
Make money
End Macro
```

so the If/Else test in the first five lines sends macro execution to either of two parts, Party or Work. The comment lines of equals signs just make the script easier to read.

In our actual script, below, we're using one label, "top." WP goes right by it when starting, and proceeds to search for double hard returns, as before. If it doesn't find any (FindStatusFlag=0), the macro ends, resetting the Find/Change options to the defaults. Otherwise, it positions the cursor after the double returns, and selects that paragraph. Within the selection, it then searches for a colon. If it finds one, it extends the selection – meaning that the selection now reaches from after the double returns to the colon – and it makes that selection bold.

If it doesn't find a colon, it moves the cursor left one character to deselect the paragraph with the cursor at the start of it, so it will be able to see the double returns following that paragraph. Then, with the Go command, execution goes back to the top of the script to start over.

Compare this script with the flowchart in figure 12 and see how they match up.

```
Label (top)
;=====================================================
; ready to search for the next double hard returns
;=====================================================
Find/Change Direction (Forward;No Wrap)
Find/Change Where ({Current Doc})
Find/Change Match (Partial Word;Case Insensitive;Alphabet Insensitive;CharRep
Insensitive;{Text Only})
Find/Change Action (Position After)
Find String ("[Hard Return][Hard Return]")
Find
If (FindStatusFlag=0) ; if there aren't any more double returns
     Find/Change Reset
     End Macro
End If
;=====================================================
; if we find a double hard return
;=====================================================
Select Paragraph
```

```
Find/Change Direction (Within Selection;No Wrap)
Find/Change Action (Extend Selection)
Find String (":")
Find
If (FindStatusFlag=1)
    ;=================================================
    ; if we find a colon
    ;=================================================
    Attribute (On;Bold) ; bold all text between returns and colon
Else
    Left () ; deselect text, putting the insertion point to the left
End If
Go (top)
```

Then put this script into your macro editor and try it out. Fun!

This complexity of design will seem difficult at first. Don't worry. It gets *much easier* with practice, and you'll find yourself designing macros like this to relax. Meanwhile, go through this chapter again, remember how difficult:

```
If (SelectionFlag)
    Beep
End If
```

seemed in chapter two, and realize how much you're learning!

What's going on here

Let's stop to note something important. Up to now, you've seen macro scripts, typed them in, recorded and edited them, as though transcribing some mysterious foreign language (which it was). If you've gone through the past six chapters and done the examples, though, I bet the macro scripts in this chapter have looked different to you. I bet you've *understood* the largest part of what you've seen this time, without having them explained.

Some commands, syntax and fine points won't be clear yet, of course. The thing to consider now is how fast so much has become comprehensible. And since program commands and variables, tied together with flow commands such as repeat loops, are mainly what's going on in macros, you can start to see, at this point, how you'll be able to learn the entire language (this doesn't mean memorizing it – that's what the online macro help is for) and write commands that are just what you need to work better and have more fun.

7: The Value of Example

Here's a trade secret: a vital part of learning macros is *studying examples.* Now that you've had some theory in the past few chapters, take a look at these new examples. They shouldn't be hard to figure out.

Things to record or script

As you get better at scripting, you'll find it faster to write a script than record it, at least most of the way. For example, you could record a macro to switch two letters you inadvertently typed out of order, or you could enter these lines directly into the macro editor:

```
Left (Select)
Cut
Left ()
Type Var (Clipboard)
Right ()
```

thereby bypassing WP's smart paste, which we don't want here anyway. Assign this a one-handed keystroke, and it's a real wrist-saver!

BBEdit calls it "Twiddle" and will also switch whole words. So will this macro:

```
Word Left ()
Word Left (Select)
Cut
Word Right ()
Paste
```

Here's one that you might want to assign the keystroke Option-Semicolon, since it looks much nicer than the standard ellipsis you would otherwise get with that keystroke, and doesn't interfere with word wrap:

```
Type ( .) ; space before the period
Hard Space
Type (.)
Hard Space
Type (. )
```

Note that there's a space before the period in the first line, and a space after the period in the last line.

Read-only variables

These provide information about the current state of things in WordPerfect. **LineCharacterCount**, for example, tells you how many characters there are in the line, to the left of the insertion point. That's sometimes a helpful figure to have (not to mention vital to complex macros), so you could write a macro like:

```
Alert ("There are "$LineCharacterCount$" characters to the left of the insertion point.")
```

BTW you can type these in, click Pause at the top of the macro window, bring a document window to the front and run the macro.

ChapterNumber is another. If you've set chapters (in any of the box number dialogs), so that figures and their cross-references can say "Chapter 12, Figure 16," but you don't have a figure close by to remind you of the current chapter number, you could write an alert like:

```
Alert ("The current chapter number is "$ChapterNumber$".")
```

You can of course put more than one variable in an alert. **ScreenSizeH** and **ScreenSizeV** contain the dimensions of your main monitor:

```
Alert ("Your main monitor measures "$ScreenSizeH$" points wide, and "
$ScreenSizeV$" points high.")
```

Commands with read-only variables

Want to add the Revert command that some programs have, to return to the last saved version of a file? Try this one. **CurrentDir** is the current folder, and **DocumentName** is just that.

```
Assign (Var00;CurrentDir)
Assign (Var01;DocumentName)
Close
Open Document (Var00$Var01)
```

Note that the **Close** command closes the active file *without* saving any changes! Put the **Save** command first in a script where you want to save changes.

I use something similar: I often have more than one version of a file on disk, and want to make sure that the one I opened with Now Menus' open recent menu is the one I think it is. So I run:

```
Alert ("Active file is "$CurrentDir$DocumentName)
```

although if you've opened a file in another folder (and then closed it, or sent it to the back) since opening your active file, the path you see may be the one to the more recently opened file.

Here's another variable. It's nice to use WP's Next Window command on the Window menu to cycle through all open windows. I often have four or five windows open, though, and often want to go from the top one to the one right behind it, and back again. So I run this macro (using a convenient one-handed key assignment), with the variable **NextWindow**:

```
Select Window (NextWindow)
```

except that this returns an error if there's only one window open.

Conditional statements

No problem; let's add the variable **NumberOfWindows**, and get:

```
If (NumberOfWindows>1)
```

```
    Select Window (NextWindow)
End If
```

which is also an example of a conditional statement. An elaboration of that would be:

```
If (NumberOfWindows>1)
    Select Window (NextWindow)
Else
    Beep
End If
```

to let the user know when there isn't a next window to be selected.

If you typically work with different font sizes, including small ones, and would like to save a step when choosing screen magnification, try:

```
If (FontSize<10)
    Magnification (150)
Else
    Magnification (100)
End If
```

except that this doesn't have many options. But we don't need to limit ourselves to one **If** condition.

Adding an operator

We can use the ampersand as the **and** operator, toss in another Else condition, and get:

```
If (FontSize<10)
    Magnification (150)
Else
    If (FontSize>=10&FontSize<18) ; font size between 10 and 17
        Magnification (100)
    Else
        If (FontSize>=18)
            Magnification (75)
        End If
    End If
End If
```

so that the fourth line specifies that if the read-only variable FontSize contains a value (for the font size at the insertion point) equal to or greater than 10 *and* a value less than 18, set magnification to 100%. Edit this macro for the font sizes and magnifications you want, give it a convenient keystroke, and get used to more legible working conditions.

This is also a good example of nested commands: when you type this in your macro editor, commands following both If and Else are indented, with the multiple End If lines ending the indents. This tells both the macro editor and you what, at any point, the If and Else considerations are.

Repeat loops

We looked last chapter at a repeat loop. Here's another example of that, with the read-only variable **FrontWindow**, containing the name of the active document, and the commands **Cycle Windows** and **Print**. The Print command here contains the parameter **Document**, meaning print one copy with the print options currently in effect.

As the Cycle Windows command repeats, the value of the FrontWindow variable will change. Var01 continues to contain the name of the original active document and, when those two are equal again, the macro stops. So this handy macro prints all open documents:

```
Assign (Var01;FrontWindow)
Repeat
    Cycle Windows
    Print (Document)
Until (Var01=FrontWindow)
```

Here's one that changes all soft returns in your file to hard returns:

```
Repeat
    Find Next Code (Forward;Return-Soft)
    Hard Return
Until (FindStatusFlag=0)
```

A slightly shorter way to write the last line is:

```
Until (!FindStatusFlag)
```

which uses an exclamation point, the **not** operator. It's the same thing to specify that the FindStatusFlag equal the number zero, or not exist.

8: Menus

So far, we've written and implemented basic commands that are fairly linear: press a keystroke, get an action. Let's take a big jump up from this, and write a macro that gives the user a choice, expressed on a **menu**.

Basic menus

To show or hide white space (margins, space for headers, footnotes etc.) more easily than going to Environment in Preferences, try these two one-liners:

```
White Space (Show)
```

and

```
White Space (Hide)
```

or, rather than have two separate macros (or 20, for another purpose), we can give the user a menu with all the options in one place:

```
Menu (Var01;"White Space";{"Show";"Hide"})
If (Var01=1)
    White Space (Show)
Else
    White Space (Hide)
End If
```

which looks like figure 13:

Figure 13: A menu

and which does this:

1. The Menu command has a variable as its first parameter: a number representing the user's choice (e.g. 1 for Show; 2 for Hide) will go into Var01. The "A" and "B" to the left of the choices are a WordPerfect extra – you can type that letter rather than use the mouse to click on your choice.

2. "White Space" is the title of the menu.

3. The words within the curly brackets form a list; in this case, they'll be the items on the menu. A list of parameters so enclosed is also called a **parameter set**.

4. The next line, with the If statement, uses the value of the variable assigned in the Menu command. If the user selected the first item on the list, Var01 would have the value 1. If the user had selected the second item, Var01 would have the value 2. But, since there are only two choices on the menu, I used an if/else structure as the simplest way to do it.

Cases and labels

If we have more than two menu choices, we could add more if/else statements or, more conveniently, tell WP that in **case** Var01 is 1, do this, and in case Var01 is 2, do that, and in case Var01 is 3, do the other thing.

For example, you can set the format orientation (in Environment in Preferences) to either Paragraph or Single Paragraph, nice options for flexibility, but you have to go into Preferences to do it. If you remember WP 1.x or 2.0.x, you know another formatting option, Character (my favorite much of the time). The difference is: say you have your insertion point in the middle of a single-spaced paragraph, and you choose double spacing from the Layout Bar. With Paragraph formatting, all lines in the current paragraph and all succeeding paragraphs (until a style takes effect) become double-spaced. With Single Paragraph, only the paragraph containing the insertion point is affected. With Character, only the text following the insertion point is affected. Ready availability of all three significantly increases power in formatting.

This macro uses the **Formatting** command with the parameters Character, Paragraph and Single Paragraph. It posts a menu to assign a value to a variable (Var01), which will be 1, 2 or 3 depending on the user's choice from the menu list. The Case command then looks at Var01 and assigns a label according to its value. Macro execution then goes to that label. (Remember, we learned about labels last chapter, using the Go command to send macro execution to one label. This is just a little slicker, having the macro branch to one label or another, depending on the value of a variable.)

The list in curly brackets in the Case command are the labels corresponding to the values of Var01. The **Cancel** label following the curly brackets is the **default label**, to which the macro will go if the user clicks the close box in the menu (that is, doesn't put anything in Var01). The End Macro command following each label and its associated command keeps the macro from going on to read and execute subsequent lines, thus resetting the formatting. This command isn't needed at the end of a script since a macro ends naturally when it runs out of code.

```
Menu (Var01;"Set Format to:";{"Character";"Paragraph";"Single Paragraph"})
Case (Var01;{1;Character;2;Paragraph;3;Single Paragraph};cancel)
Label (cancel)
End Macro
;
Label (Character)
Formatting (Character)
End Macro
;
Label (Paragraph)
Formatting (Paragraph)
End Macro
;
Label (Single Paragraph)
Formatting (Single Paragraph)
```

For an added touch of elegance, let's tell the user what the current setting is, using the read-only variable **FormatOrientation,** which holds the value 0, 1 or 2 depending on the orientation. Add these lines to the beginning of the macro, so that the last line here replaces the menu command:

```
If (FormatOrientation=0)
    Assign (Var02;"Character")
End If
If (FormatOrientation=1)
    Assign (Var02;"Paragraph")
End If
If (FormatOrientation=2)
    Assign (Var02;"Single Paragraph")
End If
Menu (Var01;"Format now: "$Var02$". Set to:";{"Character";"Paragraph";"Single Paragraph"})
```

Call

Use of menus invites sophistication in flow commands. The user's choice sends macro execution to one of several different labels, which is fine as long as all those labels do different things. Sometimes, though, they contain some identical steps. Rather than repeat those in each label, it's easier to refer execution to another part of the script, run the commands there, and then return. Here's a quasi-macro example:

```
Menu (Var01; "First Day of Vacation";{"Sleep All Day";"Go Fishing";"Play Golf"})
Case (Var01;{1;sleep;2;fish;3;golf};cancel)
Label (cancel)
End Macro
;==================================
Label (sleep)
Sleep
End Macro
;==================================
Label (fish)
Get Up
Wash
Brush Teeth
Get Dressed
Eat Breakfast
Put Fishing Rod in Car
```

and so on, and you can see that the next label, "golf," will include the same commands between "Get Up" and "Eat Breakfast," inclusive. In a real macro, that could be many lines of code, repeated for a dozen menu choices. A shortcut is to use the **Call** command, which will send execution to a specified label and continue until a **Return** command is encountered. Execution then returns to the line immediately after the Call command. So part of our vacation macro would look like:

```
Label (fish)
Call (firstThing)
Put Fishing Rod in Car
;===================================
;and so on
End Macro
;===================================
Label (firstThing)
Get Up
Wash
Brush Teeth
Get Dressed
Eat Breakfast
Return
```

with the "firstThing" label somewhere out of the way, perhaps at the end of the script. Execution now, assuming the user's going fishing, goes to the label "fish" and then to the label "firstthing." Upon seeing "Return," the macro goes back to the line right under the Call command, which is "Put Fishing Rod in Car." Note that the Return command is not anything like **Hard Return**, which adds a new paragraph to the document.

9: Custom Menus and Online Help

The menus we looked at in the last chapter gave you additional program commands, or made existing commands easier to reach. There's not much of a limit on what you can do with menus, though. How about your own menu, of commands, frequently-used documents, boilerplate text, anything you'd like to have in one place for fast access? Let's go.

Custom menus

First, make a list of what you want. This could be something like:

1. Append selection to clipboard
2. Open journal
3. Select entire table
4. Type your address
5. Typeover toggle

a list I got by taking a tour through the online macro help's index of commands, and seeing what I might like to put on a menu. You can take that tour, or build this menu first to see how to interpret the online help.

To make this menu, start with:

```
Menu (Var01;"My Work Menu";{"Append selection to clipboard";"Open Journal";"Select table";"Type my address";"Typeover toggle"})
```

to give you a menu title and selections. Note that all parameters for the menu command are enclosed in parentheses and separated by semicolons, and that the list of menu items is further enclosed in curly brackets, forming a parameter set.

Next, put in a Case command. This will have the same variable (Var01) as its first parameter, followed by numbers starting with 1, to reflect what the user put in Var01 with the menu and, for each number, a label. The labels can be anything you want; they just have to match the labels later in the macro.

Although this might seem largely a repeat of the list of options in the menu command above, one important difference is that the menu command has its items in quotes; the case command doesn't. So for our example, this would read:

```
Case (Var01;{1;append;2;journal;3;select;4;address;5;typeover};cancel)
```

followed by the lines

```
Label (cancel)
End Macro
```

in case the user calls the menu, and then decides to click in the close box (thereby not assigning Var01 a value). We're halfway there. All we need to do now is fill in the commands for these various labels.

Help with syntax

The online macro help will tell us how to do this. From the Balloon Help menu (not the Apple menu's Help), choose **WP Macro Help…** and from the **Help Topics** list, click **Index.** Zoom the window if you like, and scroll down to and click on **Append to Clipboard.** You'll see figure 14:

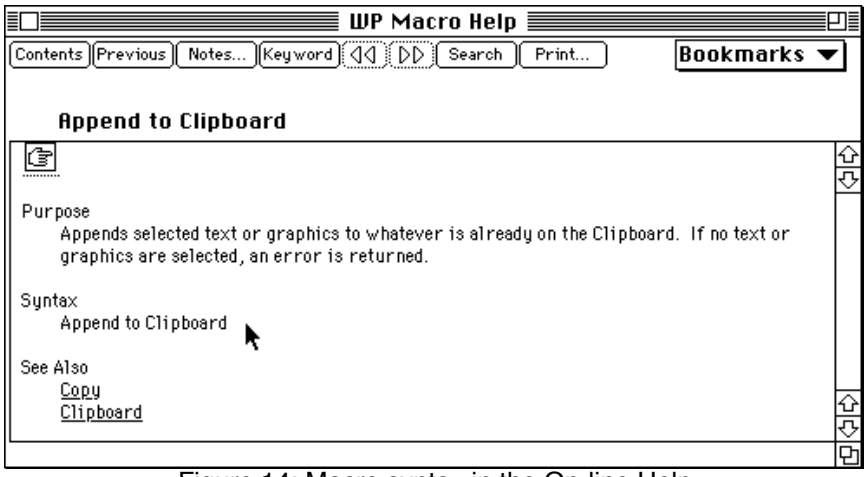


Figure 14: Macro syntax in the On-line Help

where you can click on the icon of the hand pointing right to see what program equivalent, if any, there is. You can also read the purpose of the command. The **Syntax** part is the most important for us at the moment: the command in your macro script must read exactly like this, including spaces.

So, the next lines in your macro would read:

```
Label (append)
Append to Clipboard
End Macro
```

with the End Macro command so that the macro doesn't go off and execute all the other labels to follow, as well. Note that the spelling of the label here exactly matches the spelling in the Case command.

And you get the idea, for one-line commands. But the next one, Open Journal, depends on some things: where your file is and what you've named it. No problem, just look up the syntax to see that it's like figure 15, where you can click on the dotted underlined **character expression** to see what that means. This part of your script would then read:

```
Label (journal)
```

Open Document ("Your hard disk name:folder name (if any):2nd folder name (if any, etc.):filename")
End Macro

and, of course, you could have any number of these in your menu, for documents you want to access often, but perhaps not often enough that you can depend on their being in the Open Latest menu.

You're on your own for the next label, to Select an entire Table.

The next label, to type your address, is simple enough. It's:

Label (address)
Type (66 Green Dolphin St.)
Hard Return
Type (Anywhere, USA 12345)
End Macro

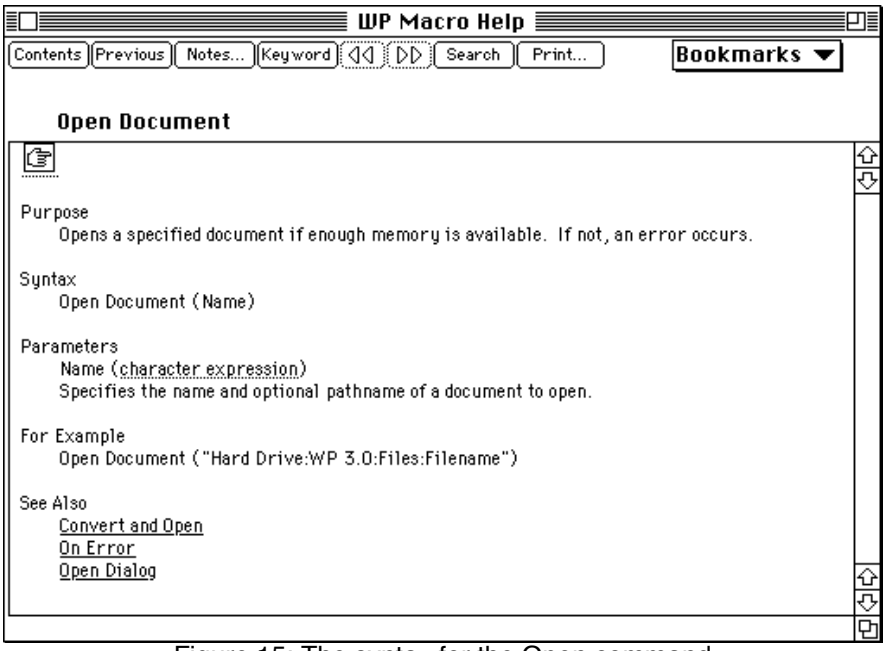


Figure 15: The syntax for the Open command

and put tabs in there (the word **Tab** on its own line, just like Hard Return) if you want your address over to the right side of the page. If you'd like, add the line:

Date Text

at the end.

And, you're on your own for the last label, Typeover. (Actually, in case you need some assistance, the whole script is at the end of this chapter.)

In case of error

This is all very nice, but remember the online macro help's description of the Append to Clipboard command? If no text is selected, this command returns an error whose descriptive alert could be more detailed. Let's **trap** that error, by an If statement that looks for lack of a selection flag, and alerts the user. So we get:

```
Label (append)
If (!SelectionFlag) ; if nothing is selected
    Alert ("You must select something to append.")
    End Macro
End If
Append to Clipboard ; the original command, if we get this far
End Macro
```

so users have more feedback about what they're doing wrong.

The error you get by choosing Select Table when the insertion point isn't in a table, is similar. You can change that alert with an If statement testing the **InTableFlag**, i.e. does the flag not exist, or equal zero? Try that.

Document management

Let's spruce up the Open Journal label. As it is, it opens the file and that's it. But it would be nicer if it opened the file, went to the end, drew a line across the page, hit return, entered the date, and hit return twice more. This would be a good part to record. With your Journal file open and the insertion point in the script editor at the end of the line with the Open Document command, make the journal file active (click in its window) and record these steps:

1. Go to the end of the file, using whatever keystroke you do that with.
2. Press Return.
3. Turn paragraph border on, top only, whatever line type you like.
4. Press Return.
5. Insert Date Text.
6. Press Return twice.

which will give you this addition to your script:

```
End ()
Hard Return
Begin Border Options (Paragraph)
Border Sides (Top On;Left Off;Bottom Off;Right Off)
Border (On)
End Border Options
Hard Return
Date Text
```

Hard Return
Hard Return

in between the Open Document and End Macro commands for the Journal label. Try that out.

Hey – this is fun

You betcha. Assign your work menu a convenient keystroke, and then see what other gems the online macro help can point you to. Look for commands you use often and would like to have more immediate access to. If there are styles you call a lot, for example, and would like to access by keystroke, you can use this syntax (from now on, instead of adding a picture of the help window, I'll enclose example syntax in a box, like this):

```
Apply Style ("Style Name")
```

or any number of text entries (although my Glossary macros are a more efficient way of doing these – we'll look at that macro in the next chapter) or opening any number of files. Another good candidate for a menu entry is any sequence you perform often, but not often enough to give it its own macro, keystroke, and spot on the main macro menu. If you find yourself taking the active document and changing the margins, spacing, font, and size, printing a copy, then changing the formatting back for further possible screen editing, put all those in a macro sequence and add it to your menu. Caution: pay attention to whether the parameter for the macro command, as defined in the online help, includes quote marks. The Font Name command does, for example, while Font Size does not.

The answers at the end of the chapter

If you ran into any difficulty building parts of this macro, here's the whole thing. The dividers, comment lines starting with semicolons, just make the code easier to read:

```
Menu (Var01;"My Work Menu";{"Append selection to clipboard";"Open Journal";"Select
table";"Type my address";"Typeover toggle"})
Case (Var01;{1;append;2;journal;3;select;4;address;5;typeover};cancel)
Label (cancel)
End Macro
; ======================================================
Label (append)
If (Clipboard="") ; if there's nothing on the clipboard
     Alert ("Can't append to an empty clipboard.")
     End Macro
End If
If (!SelectionFlag) ; if nothing is selected
     Alert ("You must select something to append.")
     End Macro
End If
Append to Clipboard ; the original command, if we get this far
End Macro
; ======================================================
Label (journal)
```

```
Open Document ("Disk:Folder:File") ; you need to edit this path
End ()
Hard Return
Begin Border Options (Paragraph)
     Border Sides (Top On;Left Off;Bottom Off;Right Off)
     Border (On)
End Border Options
Hard Return
Date Text
Hard Return
Hard Return
End Macro
; ====================================================
Label (select)
If (!InTableFlag)
     Alert ("Your insertion point must be in a table.")
     End Macro
End If
Select Table
End Macro
; ====================================================
Label (address)
Type (66 Green Dolphin St.)
Hard Return
Type (Anywhere, USA 12345)
End Macro
; ====================================================
Label (typeover)
Typeover ; this is a toggle command
```

10: Going Global

Up to now, we've used read-write variables that we wrote a value to, and then could read that value from, while the macro was running. When the macro ended, the values simply disappeared. We call these **local variables**, and they have many uses. At other times, though, we need a variable that will hold its value after a macro ends – until you quit WP or write something else to the variable. These are **global variables**. Like the local variables, there are 50 of them, denoted **GlobalVar00 – GlobalVar49**. Let's see what we can do with these.

Making a glossary

One use is to keep any piece of text until you need it. Put access to the contents of several global variables on a menu, and you have a glossary feature. I wrote one of these, with a menu giving you up to 26 glossary entries, each up to 255 characters (the maximum string length a WP variable can hold).

Aside from the length limit, there's quite an advantage to putting a glossary entry in a variable rather than putting the actual text in the macro script, as we did last chapter, with an address on a work menu. Simply, the entry is much easier to change if we put it in some reference document, which a macro could find and read. You could of course change the text in the script itself, but as you become more advanced with macros you'll find yourself writing them for associates as well as yourself and, if your associate isn't learning macros too, asking him/her to edit a script is a daunting task. So let's do it better.

For a start, create a new document with a table of one column and 26 rows. Name this document "Glossary File" and save it in the WordPerfect folder in the Preferences folder in your System Folder (or leave the Preferences folder out of that path if you're using system 6). Put a few words of text in the first cell of that table, and close the file.

Now we want to cook up a macro that will open this file (probably when you start WP), and put the contents of that cell into a global variable. A second macro, which we'll run when we want to insert the glossary entry, will put the contents of that variable on a menu, and the user can just click on it. The macro will then type the variable into the document at the insertion point.

The macro to assign entries

So the first macro starts by opening the glossary file you just made. How does the macro find it? When you start WP, one of the things it does is ask your Mac for the location of its system folder, and then puts that into a read-only variable called **BootDir**. So we can start the path with that, and not worry about what you or anyone else has named their hard disk, or how many levels deep the System Folder is, and so on. The first command is then:

```
Open Document (BootDir$"Preferences:WordPerfect:Glossary File")
```

with the join operator ($) hooking up the read-only variable with the character expression containing the rest of the path to the file. As a character expression, it's in quotes.

With the file open, our next step is to put the insertion point in the first table in the document (here, the only table) and in column 1, row 1. That command is: **Position to Cell**, whose syntax is:

Position to Cell (Table ID;Column;Row)

where the **TableID** parameter is the number of the table in the document. Note that the variable has no spaces in its name, by convention.

And we get:

Position to Cell (1;1;1)

putting the insertion point in the first cell. From there we can :

1. select the contents of the cell
2. copy
3. assign a global variable to the contents of the clipboard

and, since people generally start using variables 1, 2, 3 etc. in their macros, let's start using globals from the other end, the better to stay out of another macro's way. Since our globals are going to stay around as long as WP is running, the other guy's globals will too. So we'll start with GlobalVar49. Make a new macro entitled "Assign Glossary Entries" (no need for a keystroke), and put this script in it:

```
Select TableCell
Copy
Assign (GlobalVar49;Clipboard)
Close
```

We'll add to this later, but all we'll need to add are repetitions for more glossary entries, to be contained in more globals.

A menu for global variables

Our next macro will put the globals (the one we have now, and more later) on a menu. A case command will follow, to send macro execution to the proper label. Make another macro, call it "Glossary," and assign it a keystroke you like. Start it off with this script:

```
Menu (Var01;"Glossary";{GlobalVar49})
Case (Var01;{1;49};cancel)
Label (cancel)
End Macro
Label (49)
Type Var (GlobalVar49)
```

and let's look at a few points. First, we're using a local variable, Var01, for the menu and case commands. Why not? We can use both local and global variables within one macro. We'll have no need for the contents of Var01 after this macro ends, so a local variable is a good place for it.

Second, there's an important command here: **Type Var** will put the contents of any variable into the active document at the insertion point. Not a bad macro command for a word processor to have.

Third, we've written menus with character expressions (i.e. regular text), as in:

```
Menu (Var01;"Menu Title";{"Menu Choice 1";"Menu Choice 2"})
```

where text has to be in quotes. The Glossary menu above uses GlobalVar49 as a menu choice, and it's not in quotes. If it were, you'd see, literally, "GlobalVar49" on the menu – probably not much help. But the *contents* of a variable can be used in place of a character expression, where the variable replaces the expression and the quotes marking it as such. Think of the variable containing text the same way that quote marks contain text.

The point of doing the menu this way is greatly expanded WYSIWYG. This menu won't say something like "Glossary Entry 1" but will show the actual text written to the variable, or as much of it as will fit on the menu.

Try these macros out: run the Assign macro first, then the Glossary macro. You see the potential, with more entries. You also see the potential for error, if the user runs Glossary without running Assign first. One way around that, for the users who need a glossary feature on a regular basis, is to have their **OnStartUp** macro run the Assign macro. OnStartup runs whenever you start WP (and **OnOpenDocument** runs whenever you open a document containing it or, if it's in the Library, whenever you open any document), so a line in OnStartup could be:

```
Run ("Assign Glossary Entries")
```

which is one way to run one macro from another.

But our user may not want to add that line to OnStartup. Well, in any case we can have the Glossary macro check to see if Assign has been run – that's to say, see whether it's put anything in GlobalVar49. Put these lines at the top of Glossary:

```
If (GlobalVar49="")
    Run ("Assign Glossary Entries")
End If
```

and the structure of things is largely finished. All we need to do now is add the code for multiple entries. As you might expect, this will be fairly repetitive, as is a lot of code. Copy and Paste are good friends when writing a comprehensive script.

For the Assign macro, though, we need to tell the macro how to go from one glossary entry (that is, one cell in the table) to the next. What command would we use? "Down ()"? Nope. It moves the insertion point down a line, which may be in the same cell for a multi-line glossary entry. Here's a good place to click the **Pause** button at the top of the script editor window, and go back into the program to see what it does. Play with a table for a minute and you'll see that **Tab** is a good keystroke to go from one table cell to the next. To get back to the macro editor, just bring its window to the front. You can click the **Continue** button if you want. Clicking Pause in the first place just keeps the macro from recording your actions when any other window is in front.

As a nice touch for the user, you can add a **prompt** while assigning entries. This line, at the top of the Assign macro, would be something like:

```
Prompt (75;125;"Glossary";"Reading glossary information . . .")
```

where the first and second parameters are the distance in pixels/points from the top left of the screen to the top left of the prompt. The third parameter is the title, and the fourth is the text.

Put an **End Prompt** command at the end of the Assign macro, so as not to leave the prompt on screen.

Conceptual point: in past chapters I've talked about recording part of a macro and scripting the rest. Here, though, we've scripted part of things and then gone back to the program not to record but just to see how things work. We then take that observation and write our script to fit.

11: Windows

No, not the Microsoft kind, thank you, but those things on our Mac screen. Let's look at how macros can make our use and handling of windows easier and more fun.

Tiling windows horizontally

The stock WP command to tile windows sets two open windows next to each other vertically, and tiles larger numbers of windows to approximate squares on screen. Although this arrangement is useful for an idea of what each window contains, I work better with windows tiled horizontally so that each one shows whole lines of text. Sizing and moving windows by hand is a drag, though (pun intended), so let's write a macro to do it.

First off, what kind of information do we need? To calculate window sizes and positions so that each of, say, four open windows takes up one-fourth of the screen (minus the menu bar), positioned succeeding fourths of the way down the screen, we'll need to know the screen size. This is provided by the two variables **ScreenSizeH** and **ScreenSizeV.** We'll also need the number of open windows, for which the variable is **NumberOfWindows**. How did I learn about these variables? I wandered around the online macro help, thinking as I looked at each command and variable, "What can I do with this one?"

Off we go. ScreenSizeH we can use as is, since there's nothing else taking up screen space horizontally. This will give us windows at the width of the screen. We'll have a fix for that later, to give us windows at standard width.

For the vertical dimension, we'll be sizing windows so that if two windows are open, each takes up half the screen; if three, a third and so on. The vertical dimension will thus approximately be **ScreenSizeV/NumberofWindows** – approximately, since we have to subtract the 20 points (or pixels) taken up by the title bar, since WP doesn't include that in measuring window size.

The commands we'll use with this data are **Size Window** and **Move Window**. These both take the name of the window as their first argument, as you see in the online macro help. You could manipulate specific windows this way: "Size Window ('My First Novel')" etc., or use the variable **FrontWindow** to specify that you want to operate on the frontmost window.

So we start with this code:

```
Size Window (FrontWindow;ScreenSizeH;(ScreenSizeV-20)/NumberOfWindows-20)
```

and note that for the vertical dimension, we're dividing the height of the screen by the number of windows minus 20 (for the size of the usable screen) and further subtracting 20 so the windows will fit in the usable screen. Units of measurement in macros are always points unless otherwise specified.

Once the front window is sized correctly, let's move it, with the Move Window command. Since you're looking up the syntax of each command in the online help as we go along (aren't you?), you see that the arguments for this command are Window Name; Horizontal Position; Vertical Position. So we get the line:

```
Move Window (FrontWindow;1;(ScreenSizeV-20)/NumberOfWindows*Var01+40)
```

Hold it. What's "Var01+40" doing in there? Here's where things get a little complicated conceptually. All windows are going to be the same size, but at different positions. What we need to do is keep count of the number of windows as we move and size them, so that we put the first window in the top position, the second window in the second and so on. Let's write a counter:

```
Assign (Var01;Var01+1)
```

so that every time we repeat the procedure, Var01 will tell us which window we're working with. The addition of 40 corrects for the Menu and Title Bar, each 20 points high.

So that this procedure cycles through all open windows, and repeats the right number of times, we'll specify a repeat loop, and add these lines:

```
Repeat
    Cycle Windows
    ; Rest of script here
Until (Var01=NumberOfWindows)
```

and we're almost there. For simplification, we worked this up leaving the horizontal dimension alone, so the script now sizes windows to the width of the screen. This would be better for some monitor sizes than others. Generally, I like my windows at standard width. We could simply specify that width (630 points), except that it's a little wider than the screens on compact Macs. So, at the start of the script, let's get the screen size, compare it to 630 points, and go with the smaller of the two, as in:

```
If (ScreenSizeH<630)
    Assign (Var00;ScreenSizeH)
Else
    Assign (Var00;630)
End If
```

and use Var00 instead of ScreenSizeH in the rest of the script. For a final touch, let's turn display off with the line **Display (**Off**)**, since the macro will run much faster if it doesn't redraw the contents of each window as it goes along.

Here's the whole thing:

```
Display (Off)
If (ScreenSizeH<630)
    Assign (Var00;ScreenSizeH)
Else
    Assign (Var00;630)
End If
;
Repeat
    Cycle Windows
    Size Window (FrontWindow;Var00;(ScreenSizeV-20)/NumberOfWindows-20)
    Move Window (FrontWindow;1;(ScreenSizeV-20)/NumberOfWindows*Var01+40)
    Assign (Var01;Var01+1)
Until (Var01=NumberOfWindows)
```

which will work for any number of windows, until the number that, depending on your monitor, would require each window to be smaller vertically than WP's minimum size (try making a

window as small vertically as you can, and you'll see what that is). So if you run this macro with 16 windows open, they're going to fall over each other. With a more reasonable number, they'll line up perfectly.

The current version of these (see Appendix D) is Button Bar-aware. With window sizing and placement, working with the Button Bar is much like the Menu.

New view

If you've used spreadsheets, you know about split windows: open a second view of the same document, which you can scroll independently and see whether what you're saying on page 25 jibes with what you said on page 11. WP will let you open a read-only second window of an open document, and you can then size and move the two views so you can see them both. If they're the only two windows open, you can run the Tile Horizontally macro we just wrote. But what if you have five windows open, but want a second view of the active document, so that these two views take up the entire screen? Well, we can do that too.

This macro will have similarities to Tile Windows. The main difference is that it will open a file, using the command **Open Document**. This command can specify a file to open, as in:

```
Open Document ("Some Disk:Some Folder:Some File")
```

with the path in quotes, and parts separated by colons. The command will find this file as long as there is one by that name in that location. If not, the macro returns an error and quits.

In the present case, we don't know what the document name or location will be, just that it's the active file. No problem: the read-only variable **DocumentName** tells us the file's name. So we can start with:

```
Open Document (DocumentName)
```

which we'll follow with commands to move and size windows – only two this time – to fill the screen. We can handle the first window with:

```
Size Window (FrontWindow;Var01;ScreenSizeV/2-40)
Move Window (FrontWindow;1;ScreenSizeV/2+40)
```

so that the original window of the document fills the top half of the screen. The next step is harder, though, since we don't know how many windows the user has open. If he or she had only one window open, and we just opened a second, we could just Cycle Windows and perform moving and sizing on the new active window. What if the user has three other windows open, though? The design we want will open a new view of the active doc and set both views to take up half the screen, in front of all other windows. The Cycle Windows macro command (just like the Next Window program command, on the Window menu) brings the back window to the front, while what we want to work with is the two frontmost windows. So using the Cycle Windows command once isn't going to work.

Instead, let's use:

```
Assign (Var49;FrontWindow)
Repeat
    Cycle Windows
Until (Var49=FrontWindow)
```

which sets Var49 to the *name* of the (new) front window, and then keeps sending the back window to the front, until the current front window has the *same* name as Var49. This won't be the new window, though: it will be the original front window.

At this point we can size and move the original window:

```
Size Window (FrontWindow;Var01;ScreenSizeV/2-20)
Move Window (FrontWindow;1;40)
```

so that the original, editable window is at the top, and we're done! Try this, and then put a few lines of code at the top of the script to set window width to standard or to screen size, whichever is less.

Note that this macro won't automate the "File is already open for writing – open as read-only?" dialog. What we can fix, though. is the error message we'd otherwise get if the user clicks Cancel in that dialog. We do that with a first line:

```
On Error (end)
```

and a last line:

```
Label (end)
```

so if the user clicks Cancel, the macro goes to the label specified if it returns an error – and just ends without further dialogs. Here's the whole script:

```
On Error (end)
If (ScreenSizeH<627)
    Assign (Var01;ScreenSizeH)
Else
    Assign (Var01;627)
End If
Open Document (DocumentName)
Size Window (FrontWindow;Var01;ScreenSizeV/2-40)
Move Window (FrontWindow;1;ScreenSizeV/2+40)
Assign (Var49;FrontWindow)
Repeat
    Cycle Windows
Until (Var49=FrontWindow)
Size Window (FrontWindow;Var01;ScreenSizeV/2-20)
Move Window (FrontWindow;1;40)
Cycle Windows
Label (end)
```

Second window

To finish up, let's look at the macro I probably use more than any other. Even though I may have three or more windows open, I go from the top window to the second window and back more often than I visit the others. I've assigned a one-handed keystroke to this macro for maximum convenience.

This macro could be a one-liner:

```
Select Window (NextWindow)
```

and that would be it. If there were only one window open, though, and you called this macro, it would return an error. So an elaboration is:

```
If (NumberOfWindows>1)
    Select Window (NextWindow)
End If
```

and, if you wanted a beep when there's only one window open, you could write:

```
If (NumberOfWindows>1)
    Select Window (NextWindow)
Else
    Beep
End If
```

But if you liked to work with two views of the same window, so the string in the variable FrontWindow has the same value as the string in NextWindow, you haven't gotten those two windows to switch yet. Try this:

```
If (NumberOfWindows>1)
    If (FrontWindow=NextWindow)
        Assign (Var01;NextWindow)
        Repeat
            Cycle Windows
        Until (Var01=FrontWindow)
    Else
        Select Window (NextWindow)
    End if
Else
    Beep
End If
Display (On)
```

12: Substrings and Things

There are various ways to get information from the user, aside from menus and bribery. We looked at Get Integer recently; another is **Get String**. We'll look at the difference between the two in a minute. First, let's code a simple macro to navigate within a table.

We'll use Get Integer to see what row the user wants to go to. Looking in the online help, we remember that the syntax for this command is:

```
Get Integer (Variable;Lower Limit;Upper Limit;Title;Prompt)
```

Since the upper limit for table rows is 32,767, let's use that, and start with the line:

Get Integer (Var01;1;32767;"Go to Row";"Enter row in table to go to:"**)**

The second line, to go to whatever row in column one, is:

Position To Cell (TableID;1;**Var01)**

Note that the read-only variable TableID is used as a command parameter, telling WP to position the insertion point within the current table.

Give that a try, with your insertion point in a table. This is basic, without provision for errors such as the insertion point's not being in a table, or the user's entering a larger number than the table has rows. We'll fix these. First, though, let's add to the macro so that the user can choose both column and row to navigate to.

More complex data

That's not as easy as it might seem, since the WP **Get** macro commands only have a place for one entry. We could put a second Get Integer command in the macro, to specify columns, but it doesn't seem very elegant to have one dialog box followed immediately by another, after which something happens. We could instead ask the user to enter both column and row number in one dialog box, with the two figures separated by a character such as a slash.

Of course, something like "12/24" isn't an integer any more. All the computer could see it as, is just a sequence of characters, which we call a **string**. Any time you use the Find/Change dialog, you're searching for, and maybe replacing, strings. A string could be all numerals; you could search for the string "123" just as easily as anything else. But, defined as a string, you couldn't add "123" to a number.

Let's expand on these definitions:

String: any sequence of any length of letters and/or numerals. Examples: "123"; "one two three"; "23-skiddoo." The first example, "123," may look like a number to you but, to the computer, it may be a number or a string, depending on where it came from. Copied to the clipboard, it's a string. This is because the Mac just takes whatever you select and copy, and calls it a string instead of figuring out whether it would make sense as a number.

Number: something that looks like "123" but which, as an actual number, can be added to, subtracted from, or used in other arithmetic operations. "23-Skiddoo" could not be a number in any case; it just doesn't make sense as a number. An integer, of course, is simply a whole number, positive or negative, or zero.

In scripts, generally speaking, strings are enclosed in quote marks while numbers are not. That's why, earlier on, we learned to format with lines like:

```
Font Name ("Palatino")
Font Size (12)
```

So if you had a font *named* "12" it would be in quotes in the Font Name command.

Let's add two more concepts:

Substring: part of a string. The string "12" is part of the string "123" but not part of the number 123 – no string is any part of any number, since they're apples and oranges. Also, the number 12 isn't part of the number 123 – it's just a smaller number. A smaller number is in a manner of speaking part of a larger number, but 45 is then also part of 123.

Substring Position: "1" is a substring of "123" in position 1. "3" is in position 3. In the string "Bebop," "op" is a substring in position 4.

So what we'll do with all this is take the user's input: "12/24" and find the substring position of the slash. It may be the second or third character in the string, and we need to know which. We'll then get the substring going from the first character to the character before the slash and, finally, the substring consisting of the first character after the slash, going to the last character. To find that last character, we'll get the **string length** of the user's input. These concepts are shown graphically in figure 16. Note that strings are shown in quotes; numbers are not in quotes. That's how we want to think about them.

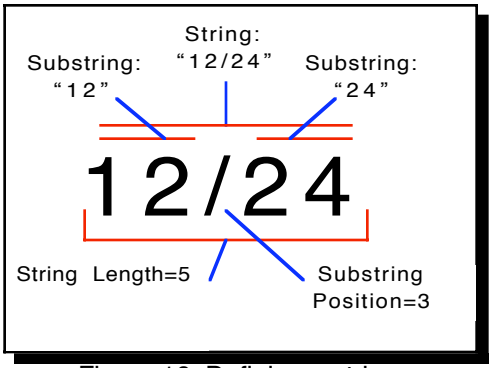


Figure 16: Defining a string

So let's start with the **Get String** command. The syntax is:

```
Get String (Variable;Maximum Length;Title;Prompt)
```

with a maximum length parameter just as a check. The largest input we'd expect, with the maximum of 32 columns and 32767 rows to a WP table, would be 8, consisting of 7 numerals plus the slash. So start a new macro, call it "Go to Table Cell" and enter:

```
Get String (Var01;8;"Go to Column/Row";"Enter column and row you want to go to, separated by a slash:")
```

and we'll get some data to work with in Var01.

On a side but important note: now that you're a WP macro programmer, you're an interface designer too. If all the macros you write are for your own use, this is less important, but other users might not know beforehand that your table navigation macro will ask for column-slash-row. So the dialog box we wrote shows them that, by example in the title, and description in the prompt. A little forethought here is much better than all the questions you'd get otherwise.

Now that we have this data in Var01, let's find how long it is with the command:

```
String Length (Var02;Var01)
```

and get the position of the slash with the command:

```
SubString Position (Var03;"/";Var01)
```

which would give us, for our example "12/24", a string length in Var02 of 5, and a substring position for the slash, in Var03, of 3.

To get the column data, we use the Substring command which, as the trusty online help shows us, has the syntax:

```
Substring (Variable;Start Position;Length;Character Expression)
```

where a character expression could be a word in quotes, or a variable not in quotes, to give us:

```
SubString (Var04;1;Var03-1;Var01)
```

so that the column data we want is going into Var04. It starts with the first character of Var01 and extends to the position of the slash minus 1.

I know this is hard at first. You might want to go over the command syntax for String Length, Substring Position and Substring again. Then, it should be easier to get the row number the user wants with:

```
SubString (Var05;Var03+1;Var02;Var01)
```

and we're almost there. Almost, because the substring commands have operated on strings with results that are themselves strings. But WP can't count a *number* of columns or rows with a *string.* So we add the commands:

```
String To Number (Var04;Var04)
String To Number (Var05;Var05)
```

to convert (programmers say **coerce**) the strings to numbers. Here I'm replacing a string in a variable with a number in that same variable. Had I wanted to keep the string for future use, I would have typed:

```
String To Number (Var06;Var04)
```

to have both to work with: the number in Var06, leaving the string in Var04.

Now, all we need is the line:

```
Position To Cell (TableID;Var04;Var05)
```

and we have a working macro! Try this out, with your insertion point in a table, and entering column and row numbers that aren't bigger than the table you're in.

Note: if you run a macro and get an error message saying: "Macro Terminated: Error reading parameter for this command," you may be mixing data types – strings and numbers.

Cleanup

Unfortunately, you can't give that last sentence's worth of advice to your users. To do so, you'd have to write a "manual" and, as you know, Mac users don't read manuals. So let's trap the errors the user might make. As a quiz, I'll give you the snippets of code, and you figure out where to plug them in (although the whole script is at the end of the chapter).

To start with, put the label "end" as the last line in your script. We'll send the user there if he or she does something wrong.

To check that the insertion point is in a table, add:

```
If (!InTableFlag)
    Alert ("Your insertion point must be in a table.")
    Go (end)
End If
```

To check that the user doesn't enter data that's bigger than the table, use the read-only variables **TableMaxColumnNum** and **TableMaxRowNum** for the lines:

```
If (Var04>TableMaxColumnNum|Var05>TableMaxRowNum)
    Alert ("This table isn't that big.")
    Go (end)
End If
```

with a new operator in the first line. The "|" symbol, which you get by typing shift-backslash, is the logical **or** operator: this line says that if Var04 is greater than the MaxColumn variable *or* Var05 is bigger than the MaxRow variable, post the alert. This logical operator is an inclusive use of "or"; we might say "and/or." If either Var04 or Var05 or both are bigger than the table, you'll see the alert.

The user might also forget to enter the slash. To check for that, add the lines:

```
If (Var03=0)
    Alert ("I can't find a slash in your entry.")
    Go (end)
End If
```

since Var03 contains the position of the slash in the string the user entered.

The next snippet isn't an error check, but a convenience for the user. As written so far, the macro moves the insertion point to the desired cell. In a big table, the user might still have to search for it. Why not select the cell once we're there, with the command **Select Tablecell**? But when testing that, I found that if I ran the macro, ending up with a selected cell, and then ran the macro a second time with different row and column specs, the selection didn't move. So, earlier in the macro, add the lines:

```
If (SelectionFlag)
    Left ()
End If
```

simply to deselect anything that's selected. You now have a nice macro to add to your collection. Here's the entire script:

```
If (!InTableFlag)
    Alert ("Your insertion point must be in a table.")
    Go (end)
End If
If (SelectionFlag)
    Left ()
    Right ()
End If
Get String (Var01;8;"Go to Table Column/Row";"Enter column and row you want to go
to, separated by a slash:")
String Length (Var02;Var01)
SubString Position (Var03;"/";Var01)
If (Var03=0)
    Alert ("I can't find a slash in your entry.")
    Go (end)
End If
SubString (Var04;1;Var03-1;Var01)
SubString (Var05;Var03+1;Var02;Var01)
String To Number (Var04;Var04)
String To Number (Var05;Var05)
If (Var04>TableMaxColumnNum|Var05>TableMaxRowNum)
    Alert ("This table isn't that big.")
    Go (end)
End If
Position To Cell (TableID;Var04;Var05)
Select TableCell
Label (end)
```

and some special congratulations for finishing this chapter, since you're now calculating substrings and coercing strings to numbers – real programming – that is *most* impressive at parties.

13: Elementary Magic

In chapter 11 we looked at three macros that make window management easier. The one we'll look at now completes my "John's WP Window Manager" set (see Appendix D for instructions on finding this on the Internet). I've saved the last macro for this chapter because it's quite a bit more complex than the others, and uses the substring commands we learned in chapter 12. It also demonstrates a more sophisticated use of variables and a more intricate flow structure than we've seen before. There are even a couple of neat tricks.

What this macro does is help the user manage the number of windows he or she has open. Cycling through all open windows, it asks the user whether or not to save and close each window, close without saving, leave open, or close all windows (asking to save changes in each one). When all windows have been considered, the macro ends.

Let's look at the macro's menus and dialog boxes. The main menu, in figure 17, is the first the user sees if the active document has been modified since being saved:

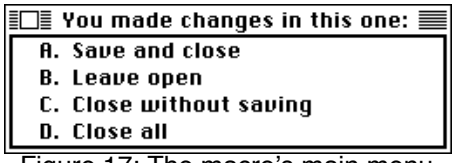


Figure 17: The macro's main menu

while if the front window has not been modified, the user sees figure 18:

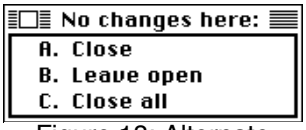


Figure 18: Alternate main menu

If the user closes the window, saving if appropriate, or leaves it open, the macro goes to the next open window and repeats. If the user opts to close all windows, the macro goes through each window, posting figure 19 if the document has been modified.:

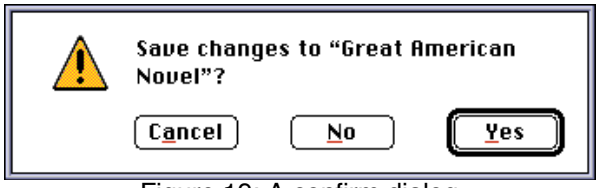


Figure 19: A confirm dialog

By the way, you see underlined letters in buttons in my screen shots of dialog boxes because I use a free control panel called "Keys!" which you can download from several Macintosh archives.

Type the underlined letter to press the button, without using the Command key. Try it – it's slick. While you're there, check out my other macro sets as well. I'll look forward to seeing *your* ideas there before too long.

Simplified flowchart of the thing

Back to the topic: figure 20 is an overview, in the form of a flowchart, of the macro:

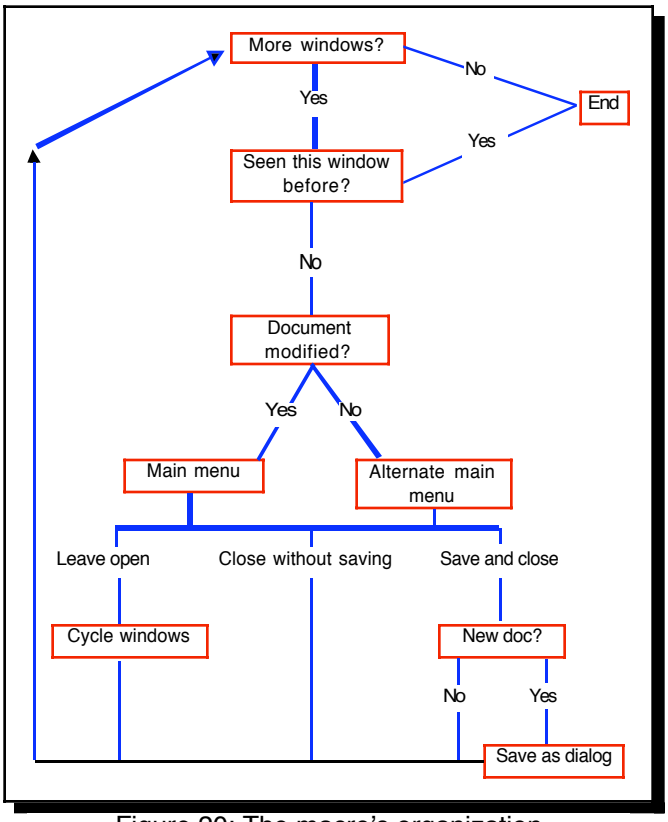


Figure 20: The macro's organization

This is a fairly complex structure, necessary to give the user this many choices. This flowchart leaves out the Close All option, which just repeats the Save and Close/Close without Saving choice for each remaining window. As you'll see, writing complex scripts becomes easier if you flowchart what you want to do, then write small pieces, and then put them together.

Get your rabbit and hat

Much of the code is straightforward enough; the tricks come in where we need to tell, as we're cycling through windows, if we've seen a window before. Also, we'll use some sleight-of-hand to

tell what the user does in a dialog box. This will be a good introduction to the strategy of writing more powerful macros. It's also a bunch of fun.

The only new variables we'll use are **NewDocumentFlag**, which has a value of 1 if the active document has never been saved, and **DocumentModifyFlag**, which has a value of 1 if the active document has been changed since the last time it was saved. Of course, each flag has a value of 0 otherwise.

The critical variable, though, is a local variable – I'll use Var00 – that is going to keep track of whether the next window to come along is new to the macro, or whether it's already been through the cycle, with the user opting to leave it open. If it's the latter, this is where we want the macro to stop. But we can't keep track of the windows just by counting them, as we did with Tile Windows. In that macro, there was a constant number of windows. Here, the user may be closing some windows and cycling others. We need to find a way to recognize a window by name. So, this line of code:

```
Assign (Var00;Var00$FrontWindow)
```

adds to Var00 the name of each open window the macro encounters. Say you have three windows open, called "FirstDoc," "SecondDoc" and "ThirdDoc," and you opt to leave the first one open, closing the other two. When the macro gets back around to FirstDoc and reads its name from the FrontWindow variable, Var00 will then contain "FirstDocSecondDocThirdDoc." The Substring Position command checks to see whether FrontWindow, now containing "FirstDoc," is a substring of Var00. Since it is, the macro has seen all windows and it's time for it to end. Simple, no?

So let's start with a label called "top" since, with a macro that cycles a lot, it needs a place to start over for each window. We can then check to see if any windows are still open, going to the end of the macro otherwise. We'll then check whether we've seen the current window before. If not, we'll add the name of the current window to Var00, and the first lines of code will look like:

```
Label (top)
If (NumberOfWindows=0) ; no more windows to consider
    Go (end)
Else
    ; ================================================
    ; Have we seen this window before?
    SubString Position (Var01;FrontWindow;Var00)
    ; ================================================
    ; If we find FrontWindow in the compilation of all windows
    If (!Var01=0)
        ; ============================================
        Go (end)
    End If
End If
; ====================================================
; If we get this far, add this window's name to the compilation
Assign (Var00;Var00$FrontWindow)
```

and we've done the largest part of the thinking.

Tons o' labels

At this point, the macro is looking at a window it hasn't seen before, and we want to have the script flow to one of various labels depending on whether the doc is new, whether it's been changed, and so on. The code for that is:

```
If (DocumentModifyFlag=1) ; doc has unsaved changes
    Menu (Var01;"You made changes in this one:";{"Save and close";"Leave open";"
    Close without saving";"Close all"})
    Case (Var01;{1;Save and close;2;Leave open;3;Close without saving;4;Close
    All};cancel)
Else ; doc has no unsaved changes
    Menu (Var01;"No changes here:";{"Close";"Leave open";"Close All"})
    Case (Var01;{1;Close without saving;2;Leave open;3;Close all};cancel)
End If
Label (cancel) ; user clicks in menu's close box
Go (end)
```

Notice that I used Var01 both for the substring position of FrontWindow in Var00, *and,* later in the code, for the menu/case value. Why not? With local variables, you can use them for one thing, then for another. Since there are 50 of them, I could as easily have used another one, but wanted to demonstrate this way that once the value of a variable isn't useful any more, there's no problem with using that variable for another value. The macro runs just as fast too, since assigning a value to a variable isn't a slower operation if the variable already has another value. Just make sure you're not replacing a value you'll need later on. That's why we don't want to touch Var00: its value is a reference we'll need as long as the macro runs.

In practice, it's best to use all 50 variables and then re-use those that you can, being careful not to overwrite some data you'll be using later.

If the user chooses "Save and close" from the menu, the macro will branch to these lines:

```
Label (Save and close)
If (NewDocumentFlag=0) ; doc already exists on disk
    Save
    Close
Else
    Call (NewDoc) ; go to label for docs not existing on disk
End If
Go (top) ; finished with this window, ready for the next one
```

If the user chooses the second menu option, "Leave open," the case command will branch to this label:

```
Label (Leave open)
If (NumberOfWindows=1)
    Go (end)
End If
Assign (Var02;NumberOfWindows)
Repeat
    Assign (Var02;Var02-1)
    Cycle Windows
Until (Var02=1)
Go (top)
```

which first checks to see whether only one window is open. If so, the macro has done its job, and execution goes to the end. Otherwise, it goes to the next window and sees what the user wants to do with that one. But we can't use the Cycle Windows command as is, because it cycles by bringing the back window to the front, moving all other windows back a layer. Say we have FirstDoc, SecondDoc and ThirdDoc open. We choose to leave FirstDoc open, so the macro cycles windows and puts ThirdDoc in front. If we close ThirdDoc, we're then looking at FirstDoc again, and have to once more tell the macro to leave it open before we ever get to SecondDoc.

To circumvent that, I added code to assign the number of open windows to Var02, and then repeat the Cycle Windows command and decrement Var02 each time, until Var02 equals 1, which has the effect of taking the front window and moving it to the back. Shuffling the windows that way means that when we reach a window we've seen before (and so is contained in Var00), the macro is done.

If the user chooses to close the current window without saving, the label runs:

```
Label (Close without saving)
Close
Go (top)
```

since the macro command closes a window without saving. Thus, the earlier label to Save and Close has the Save command before Close.

If the user wants to close all windows, the macro branches to:

```
Label (Close all)
If (NumberOfWindows=0)
    Go (end)
End If
If (DocumentModifyFlag=1)
    Confirm (Var03;Caution;YesNoCancel;"Save changes to ""$FrontWindow$""?")
    Case (Var03;{1;Yes;0;No})
```

which first checks to see if any windows are still open. We started the macro with that check, but you'll see in a minute why we need to make it again here. If there is a window open, we need to know whether it has unsaved changes, so we look at DocumentModifyFlag. If it's on, we use a command new to us, **Confirm**, which posts the dialog in figure 19. The first parameter is a variable for use in a case command. The second parameter designates the icon for the confirmation dialog. Figure 21 shows the available icons:

Figure 21: Note, Caution, Stop

or Generic, for no icon. If you use the Stop icon, the command beeps as well. The third parameter, "YesNoCancel" in this case, shows the buttons the dialog will have. The first button is the de-

fault, with the heavy border and which you can click by pressing Return. "OK" is the other possible button for this command, as in "OKCancel." If you press Cancel, the macro ends or goes to a label specified in an On Cancel **handler**. OK or Yes will put a value of 1 in the variable, and No gives it a value of 0.

The subsequent Case command sets up labels for Var03. If the user wants to save changes, the label is:

```
Label (Yes)
If (NewDocumentFlag=1) ; doc has never been saved
    Call (NewDoc)
    Go (Close All)
Else
    Save
    Close
    Go (Close All)
End If
```

which checks NewDocumentFlag to see whether we need to use the Save As dialog. If so, it calls the NewDoc label, as does the Save and Close option much earlier in the script. Otherwise, the macro saves and closes the document and then goes back to Close All. That's why I began Close All with a check for any open windows: once the user has decided to close all of them, the macro goes into a much smaller loop and bypasses the check at the top of the script.

If the user doesn't want to save changes in the current window and clicks No, the macro branches to this label:

```
Label (No)
Close
Go (Close All)
```

If the front window is a new document, execution branches to NewDoc, which runs:

```
Label (NewDoc)
Save As Dialog
If (NewDocumentFlag=1)
    Go (end)
Else
    Close
End If
Return
```

which posts the Save As dialog. But when I tested this, a problem arose. To write good code, you have to allow for anything the user might choose to do. If the user names and locates this new file, using the Save As dialog, and clicks the Save button to exit this dialog, everything's fine. If the user gets to the dialog and then presses Cancel, though, we have a problem since the next command is Close. What if the user wanted to leave it open? If we allow for that, by leaving out the Close command, the user then can't choose Save and Close for a new document. How can we get around this?

Consider that when we include the command for a dialog, the macro pauses until that dialog box closes. When the Save As dialog closes, the front window is no longer a new document *unless* the user clicked Cancel. So, the next line in the script again checks the NewDocumentFlag. If that flag

is on, the user must have canceled out of the dialog, and the best thing now is end the macro, and let the user decide what to do.

If the user does name and save the doc, the macro then closes the file, and returns to either of the two places in the script which call NewDoc.

Finish it off with a “Label (End),” and there you have it! Congratulations on mastering a much more complex structure than anything we’ve looked at up to now, as well as a more sophisticated use of variables. As well, the size of this chapter’s macro might have been a bit daunting. This just means, of course, that you can now design much more powerful and flexible scripts. Give yourself a pat on the back, and let’s charge ahead.

14: Fun with Files

So far we've looked at all sorts of neat stuff: variables, repeat loops, cases and labels. We'll do more of that, of course, but I want to start conceptualizing in addition to coding: how ideas for macros get off the ground to start with, and how they take shape.

Getting ideas

For example, I was wandering around Usenet the other day, and someone on comp.sys.mac.apps asked whether there were any utilities to do a global find and replace on all files in a folder. Nice idea, I thought – how can I get WordPerfect to jump through that particular hoop? (The result is in my File Manager set.)

I remembered something I saw in the online macro help called **Get File** which puts the name of the first, second, third etc. file in a given folder into variables. I didn't know if it would work, but I didn't know that it wouldn't. So I went to the online help, looked up the syntax for the Get File command, and found:

> Purpose
> Assigns the name and file type of the nth (1st, 12th, etc.) file in the current folder to specified read/write variables. Use Set Directory to change the current folder.
>
> Syntax
> Get File (File Name Variable;File Type Variable;File Number)
>
> Parameters
> File Name Variable (variable name)
> The name of any read/write variable.
>
> File Type Variable (variable name)
> The name of any read/write variable.
>
> File Number (numeric expression)
> The number of the file (in alphabetical order).
>
> For Example
> Get File (Var07;Var12;18)

so the example looks at the 18th file in a given folder, and assigns the file name to Var07 and the file type to Var12.

File types

What's a file type? The Mac Finder identifies every file (application, extension, document, prefs file etc.) by two means: the file creator (WordPerfect, Illustrator, Finale or any application that can create a file) and file type. Each program can create a file in its own proprietary type, and many programs can create files of different types. WP, for example, can create files in WordPerfect format and also in text format. Any document you create in WP has a creator of WPC2 (just four

identifying letters or numbers). If you save that document in WP format, it has the file type of WPD3 or WPD4, for program versions 3x or 3.5E. If you save it as plain text, it has the type TEXT. If you save a WP graphic by itself, it will have type PICT.

What these identifiers do is tell the Finder which application to use to open the document you double-clicked on, and tell applications whether they can read it. The drawing program Canvas, for example, knows it can read documents of type PICT, no matter who the creator, since that's a standard graphics format. Canvas can't read documents of type WPD3, though, so it won't show you WP documents in its open dialog box. You can check the file creator and type of a file by going to WP's open dialog and, from the File menu in the dialog box, choosing Info.

Putting the file type into a variable with the Get File command would be useful if, for example, you wanted to do something with all the TEXT files in a folder, but not the WPD3 files. For my Global Find/Change idea, though, I wanted to work with every file. So I knew I would be putting a variable in the Get File command line to pick up the file type, but I could just ignore it in the rest of the macro.

Back to business. Note that the online help says to use the Set Directory command to change the current folder (a directory is a folder). I thought it would be best to designate one folder in which to perform a global replace, for safety's sake as well as convenience's. But you can't write a macro for everyone's machine that sets a directory, as the online help describes:

```
Set Directory ("Hard Disk:Docs:Work Docs")
```

unless everyone's disk had the same name! But there's a way around this. WP macros have two read-only variables, **BootDir** and **WPDir**, that contain the name of and path to the System Folder (BootDir), and the folder you designated in the program preferences as the WordPerfect folder (WPDir). So I knew I could put a folder called, say, "Global Find/Change" in the WordPerfect folder, write a line like:

```
Set Directory (WPDir$"Global Find/Change")
```

and have a folder in which to put all the files I wanted to work with.

I could then have a line like:

```
Get File (Var01;Var49;1)
```

which would put the name of the first file in the folder into Var01, and the type into Var49. I used the last available variable, Var49, for the file type as a reminder to myself that this didn't contain information I wanted to use.

So far so good. Now the macro can open the file named in Var01, and we can work with it. That code is, simply:

```
Open Document (Var01)
Home ()
```

and we have the file open, ready to find and change. I added the command to go Home in case the version 3.5 user has set the Environment preference to remember cursor location, and the

cursor is in the middle of the document. We'll want to find and change from the top, just to have a place to start.

Working with a standard dialog

Now things are going to get a little tricky. I knew I could post a Get Text dialog and ask the user for text to find and to change to, but I wanted to use the standard Find/Change dialog if possible. Not only would people be more used to it, but there are all sorts of options – attributes, case and some codes – that would be more work to duplicate in a custom dialog. But we don't have macro variables that contain the contents of Find/Change. Is this going to work?

I started off with this code:

```
Set Directory (WPDir$"Global Find/Change")
Find/Change Dialog
```

and found that wouldn't do at all. I ran the macro, and nothing happened. Why? Well, back to the program to find out. I had no documents open at that point so, when I tried to open the Find/Change dialog, I found the program command (on the Edit menu) grayed out. Aha.
So I thought I'd open a dummy new document first, with:

```
Set Directory (WPDir$"Global Find/Change")
New Document
Find/Change Dialog
```

and at that point could open the first file in the folder and change all instances of the user-specified text, repeating this for the other files in the folder. This would go:

```
Get File (Var01;Var49;1)
Open Document (Var01)
Home ()
Change All
Save
Close
```

which might work. The script calls the Find/Change dialog, and opening any dialog pauses a macro until the dialog closes. If I typed find and change text in the dialog, then closed the dialog, would the macro then open the first document and Change All according to what I had put in the dialog? I couldn't stand the suspense.

Alas. I set up a folder with a test file, ran the script, entered find and change text, clicked the close box to exit the dialog, and – nothing happened. I wondered how I could get the dialog box to work. Going back to the program to test, I found that if I entered find and change text, tried to find something, and then closed the dialog, that same text "stuck" in the box when I opened it again, as long as no text in the document was selected when I opened the dialog again.

If, however, I entered text in the dialog but then clicked in the close box without clicking Find, that text didn't stick. Aha. What if I ran the macro, entered find and change text, and clicked Find (or Change All, the other active button when nothing has been found), and *then* clicked in the close box, dismissing the dialog and allowing the macro to continue? It opened the first file and found and changed as I had hoped. Eureka!

I've recounted this for you so that you won't think that people write macros in steps such as:

1. "Ahem. I think I'll write a macro."

2. Sterling code is entered into the macro editor, with Vivaldi in the background.

3. Everything works.

Would it were so, but the actual process is more a flailing about, trying this and that, wondering what will work. Sometimes you find the solution quickly; sometimes you don't; sometimes you conclude that it can't be done. Of the last alternative, sometimes you're right, and sometimes you've missed a strategy that will occur to you a week or a month from now.

The rest is easy

Once a workable strategy is implemented, however, the rest is cleanup, and we have:

```
Set Directory (WPDir$"Batch Folder")
Assign (Var02;1)
Get File (Var01;Var49;Var02)
Open Document (Var01)
Prompt (ScreenSizeH/2-240;ScreenSizeV/5;"Global Find/Change";"Enter find and
change strings, set options on menus, and click Change All, to process the active win-
dow. Then click the Close Box, and all other files in the batch folder will be processed.
Press ⌘. to cancel."); get the Command symbol (in the Chicago font) from Insert Sym-
bols
Find/Change Dialog
Save
Close
;
Display (Off)
Label (start)
Assign (Var02;Var02+1)
Set Directory (WPDir$"Batch Folder")
Get File (Var01;Var49;Var02)
If (Var01=Var03)
     Go (alert)
End If
Prompt (ScreenSizeH/2-240;ScreenSizeV/5;"Global Find/Change";'Finding and Chang-
ing in "'$Var01$'." Press Escape or  . to cancel.'    )
Open Document (Var01)
Change All
Save
Close
; ============================================
; and so forth
; ============================================
```

and we have a macro.

The rest of the code for this macro is error checking, and we can judge what we'll need there by testing the script. I found that it doesn't matter how many files there are in the folder; the Get File commands for nonexistent files don't return an error. But, if we have a line for, say,

```
Open Document (Var35)
```

when there are only 30 files in the folder, the macro chokes. Hmm. Let's get around that with:

```
If (Var35!="")
    Open Document (Var35)
    Home ()
    Change All
    Save
    Close
End If
```

so if Var35 isn't empty, the macro opens a document with that name. Otherwise, nothing. And that about does it. "About" since, remember, we opened a new document to allow access to the Find/Change dialog in the first place. So, after all the files in the folder have been processed, we still have that empty window. Let's take care of that with:

```
Select Window ("untitled*")
If (!DocumentModifyFlag)
    Close
End If
```

so that if the user has put something in that window, the macro won't just throw it away. The asterisk after "untitled" is a wildcard so, if the user has opened three new documents previously in the work session, and the doc the macro opens is thus "untitled 4," this takes care of it.

15: A Taste of AppleScript

Here's a quick look at how WP macros and **AppleScript** interface, and complement each other. Even if you decide to concentrate on one or the other, it's useful to know how the power of two works here. Besides, only your first scripting language is difficult to learn. Other languages have varying syntax, but the concepts are much the same.

WP macros and AppleScript have different strengths, though. The macros run faster, since they're entirely within WP, and are recordable, while AppleScript has the great advantage of being able to control multiple programs at once. Let's take a look at a set of macros and scripts I wrote (John's WP Citations) to make WordPerfect (3.1 and later) and FileMaker Pro (2.1v3 and later) work together to provide an academic citation environment much like EndNote and ProCite do – except my set gives you all the power and elegance of FileMaker, works within WP much as EndNote does within (cough) Word, and is free. For that matter, FileMaker costs about half what either of the dedicated reference managers does and is five times the database. Upgrade to FileMaker Pro version 3, with full relational capability, and you have ten times the database.

Apple what?

AppleScript (AS) is Apple Computer's scripting language, and includes as software the AppleScript extension, the Script Editor application, and several scripting additions, collectively called **osax** (open scripting architecture extensions), with a plural, determined after hot debate in the community, of **osaxen**. These additions are the most fun, since large numbers of very creative third-party efforts are written every day and posted here and there on the net. We'll use two osaxen, "Display Dialog" and "Beep"; both probably came with your copy of AppleScript; otherwise, both are available on the net as well.

The AppleScript extension and Script Editor come with the Mac OS and with several books on the language, and are available for download at Apple's ftp sites. Script Editor is an app that looks much like WP's macro editor. Put osaxen in a folder called "Scripting Additions" and put that in your Extensions folder in the System folder.

The very basics

As setup, have a WP window open with a few paragraphs of plain text. Then go to the Finder and double-click on Script Editor. A window will open with some stuff at the top, and an area for text editing. Enter code as shown in figure 22 (again, don't bold or indent):

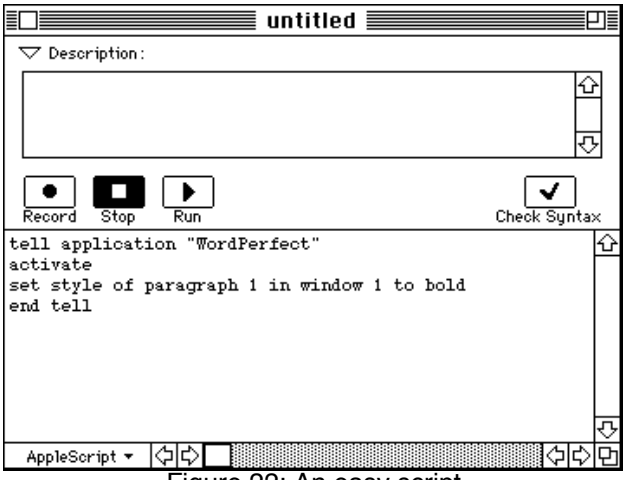


Figure 22: An easy script

Click the Check Syntax button at the right, and Script Editor will parse the text just like the WP macro editor does when you save a macro script, indenting some lines and changing some words to bold. It may also ask you where WordPerfect is on your disk, and will save this information with the script. Parsed, the text looks like:

```
tell application "WordPerfect"
    activate
    set style of paragraph 1 in window 1 to bold
end tell
```

Now click the Run button. Say hey. Just as easy as WP macros, but some different commands (choosing a paragraph by number, for example), and you can script several programs at once. Not all Mac programs, just the best ones. It will also be easy to learn, since you already know WP's language. To see how similar WP macro and AppleScript syntax are, compare the following examples. Each one does the same thing.

```
; WP macro

Get Text (Var01;" Password";"Enter password to beep Macintosh:")
If (Var01="Biff")
    Beep
    Alert ("Right on!")
Else
    Alert ("Sorry.")
End If
```

```
--AppleScript

display dialog "Enter password to beep Macintosh:" default answer ""
if result = {text returned:"Biff", button returned:"OK"} then
    beep
    display dialog "Right on!" buttons {"OK"} with icon caution default button 1
else
    display dialog "Sorry." buttons {"OK"} with icon caution default button 1
end if
```

Note that a comment line starts with a semicolon in the WP macro language, and with two hyphens in AppleScript.

See how easy AppleScript is going to be?

More magic

Let's plan some amazing strategy here. What we're going to do is:

1. With a keystroke, have WP tell AS to tell FileMaker (FM) to go to a database layout suitable for choosing a reference.

2. With a second keystroke, have FM give the relevant data to AS, which will then give it to WP, which will put it in our text, in the format we want. Caution: even as a WP Mac user, you may be startled by the degree of elegance this will have.

First off: how do WP macros and AS scripts talk to each other? Easily. To run an AS script from a WP macro, include this line:

```
Execute Apple Script ("Name of script")
```

and, if that script is on a mounted volume, WP will find it and run it. To run a WP macro from an AS script, include this line:

```
Do Script ("Name of WP Macro")
```

and presto, the two languages are talking to each other. AS gives information to WP in the form of **script variables**, which are global variables in the WP language but which are parameters in the AS command to perform a WP macro. Thus, if you have a line in your AS script like:

```
Do Script {"Name of WP Macro", "apple", "banana", "coconut"}
```

that macro will run, and **ScriptVar01** will contain "apple"; ScriptVar02 will contain "banana" and so on. There are 50 script variables, 01 through 50 (unlike globals or locals, which are numbered 00 through 49).

Off we go

As a first step, make a WP macro called "Citation" with this one line:

```
Execute Apple Script ("Get Reference")
```

and we're ready to write an AppleScript. Save and close the Citation macro, switch to Script Editor, open a new window, and enter this:

```
tell application "FileMaker Pro"
    activate -- command to bring the app to the front
    do script "All Records"
end tell
```

where "All Records," or some name like that, will be a database script that goes to a list layout suitable for choosing a record. Note that the first and last lines of this script form a **tell statement** – a common design in AS. Later we'll look at a script with two tell statements: one for FM and one for WP.
There are various ways to save an AppleScript. The first is as a compiled script, which can then be run by Script Editor or by a WP macro. Another format in which to save a script is as an **applet**, which has an application icon in the Finder, and which you can run by double-clicking. This is the most convenient for the user and, in fact, FileMaker will require us to save a script this way. So save this script as an application, using the pop-up menu at the bottom of the Save dialog, as in figure 23. Call it "Get Reference."

Note that when you choose Application from the pop-up, two check boxes will appear below it. **Stay Open** should not be checked, and **Never Show Startup Screen** should be checked.

A second script

Let's write another AppleScript to handle the data that will be coming back from FileMaker, or another database, into WP. Call this one "Put Reference." Note that there are a few syntax differences from WP's language to notice. One is in the line starting "Do Script." AppleScript won't wrap lines automatically; you need to use the symbol "¬" which you get by typing Option-Backslash on the Dvorak keyboard or Option-L on a Qwerty, to tell AppleScript that the next line is not a new command, but a continuation of the present line. For the script, though, you'll need to type a hard return following the symbol.

Another difference is that a variable can be named by any word that doesn't have a specific meaning in AppleScript. Thus, "theRecord" can be a variable just 'cause we said so.

```
tell application "FileMaker Pro"
    activate
    --"theRecord", "trans1" etc. and "Mark" are variables
    set theRecord to the current record of database 1
    set trans1 to cell "Transfer Field 1" of theRecord
    set trans2 to cell "Transfer Field 2" of theRecord
    set trans3 to cell "Transfer Field 3" of theRecord
    set trans4 to cell "Transfer Field 4" of theRecord
    set trans5 to cell "Transfer Field 5" of theRecord
    set trans6 to cell "Transfer Field 6" of theRecord
```

```
        set trans7 to cell "Transfer Field 7" of theRecord
        set trans8 to cell "Transfer Field 8" of theRecord
        set trans9 to cell "Transfer Field 9" of theRecord
        set trans10 to cell "Transfer Field 10" of theRecord
        set trans11 to cell "Transfer Field 11" of theRecord
        set trans12 to cell "Transfer Field 12" of theRecord
        set mark to cell "Mark" of theRecord
    end tell
    --action moves to WordPerfect
    tell application "WordPerfect"
        activate
        Do Script {"Enter Reference", trans1, trans2, trans3, trans4, trans5, trans6, trans7, ¬
            trans8, trans9, trans10, trans11, trans12, mark}
    end tell
```

The **set** command assigns a value to a variable. In this case, the variable "theRecord" is getting the contents of the current record in FileMaker, and "trans1" is getting the value of the database cell (intersection of field and record) of Transfer Field 1, for the current record.

The interesting part of this script is how it passes the data in those variables to WordPerfect. This is in the "Do Script" command, where "Enter Reference" is the name of a WP macro that we'll look at in the next chapter. The variables follow as parameters. WP will pick these up as Script Variables, numbered 1 through however many there are in the Do Script command, up to 50. The contents of FileMaker's cell Transfer Field 1 is thus going to land in WP as ScriptVar01. Spiffy, yes?

Save this script as an applet called "Put Reference," in a location where you won't have to move it. I put it in the WordPerfect folder in the Preferences folder.

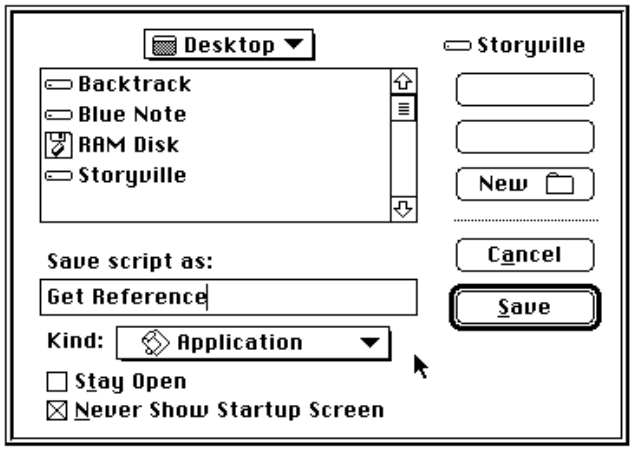


Figure 23: Saving an AppleScript as an applet

Now let's look at what you need to do with your AppleScriptable database.

Doing a database

I won't go into the details of a FileMaker script here, since you may already know its scripting language or be using a different database program. Any one will do, as long as it can calculate 12 fields worth of data and pass them to AppleScript. Typically, you might have several fields of raw bibliographic data; mine are:

Author Last Name
Author First Name
Second Author Last Name
Second Author First Name
Third Author
Translator
Paper Title
Series
Editor
Book | Journal Title
Journal Volume
Journal Number
City
Publisher
Month
Year
Orig. Date
Page Numbers
Call Number
Notes

from which I calculate 12 fields. These 12 will allow for variations in individual reference types – journal or book, for example, as well as the difference between first and subsequent citations of one reference. The calculation for one of these looks like figure 24.

A 13th field, called "Mark," which contains a flag showing whether that reference has been used previously for the current paper. The FileMaker script puts an X in that field after sending that record's data to AS. If that X is there, WP will see it and modify the next reference accordingly.

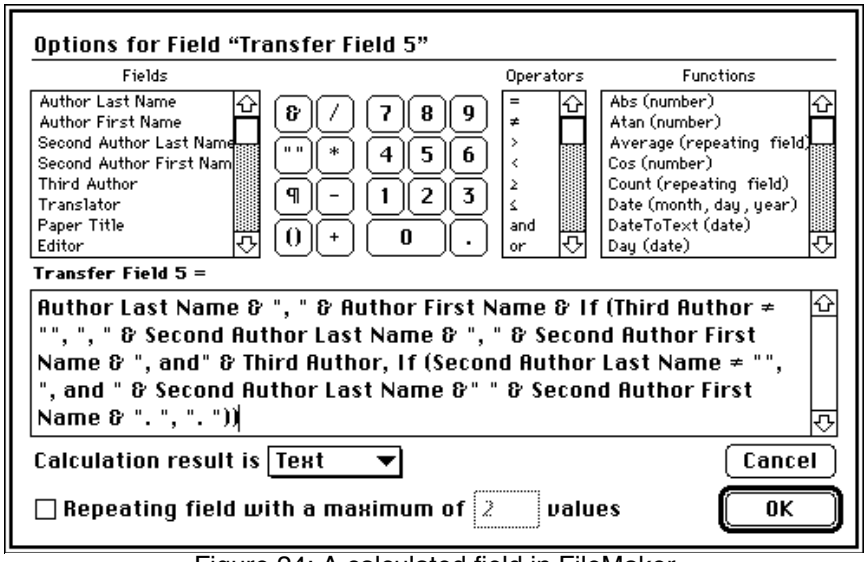


Figure 24: A calculated field in FileMaker

Using the datafile

What AppleScript has done so far is activate FileMaker or another database, and switched to a layout convenient for choosing a reference. When the user does that, just by clicking anywhere in that record, he or she can then call a FileMaker script that will call the applet "Put Reference."

This will take the transfer fields of that record and pass that data back to Applescript. To create such a FileMaker script, choose the "Send Apple Event" option in FileMaker's ScriptMaker command. You get a dialog box like figure 25, with the small popup menu near the top defaulting to "do script." Change that to "open application." The Specify Application button at the bottom will click automatically. Navigate to wherever you placed "Put Reference" and specify it. The line in the FileMaker script will then be:

```
Send Apple Event ["aevt", "oapp", "Put Reference"]
```

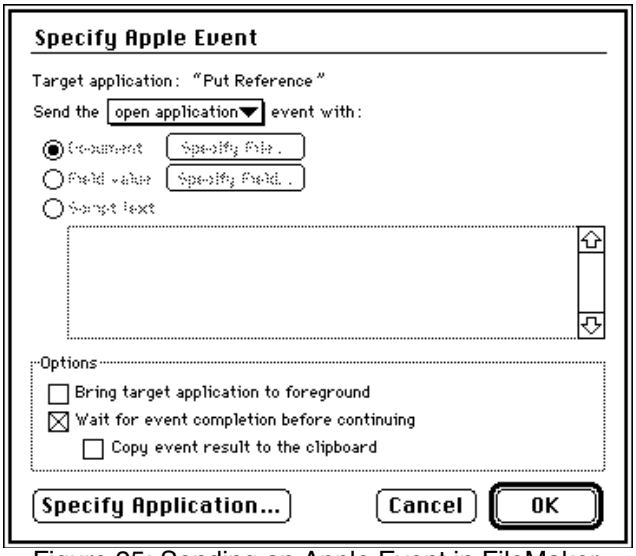


Figure 25: Sending an Apple Event in FileMaker

and we're done with AppleScript and FileMaker! Next chapter we'll study an elegant WP macro that takes these script variables and puts them into the user's choice of in-text citation, footnotes, or endnotes, or any of those plus a bibliography. The user will also be able to set-and-forget the format of each of the 12 fields to bold, italic or underline.

16: Reference Manager, part two

In the last chapter we looked at the AppleScript and FileMaker Pro (or equivalent database) parts of this citation engine. We told WP to tell AppleScript to activate FileMaker and run a script that set up the database for the user to choose a reference. FileMaker then passed the data of that reference back to AppleScript, which in turn gave it to WordPerfect.

So we're now back in the land of WP macros, and have 13 script variables worth of data. Script variables receive information from an AppleScript and then, in the WP macro environment, behave just like regular global variables: as we saw in chapter 10, they retain their value as long as WP is running, or until assigned another value.

As presently set up, the script variables contain this data:

ScriptVar01: author's or authors' last name(s), date
ScriptVar02: page numbers
ScriptVar03: blank
ScriptVar04: blank
ScriptVar05: all authors' full names
ScriptVar06: paper title, if any, and editor, if any
ScriptVar07: book or journal title and series, if any
ScriptVar08: journal volume and number, if any
ScriptVar09: city of publication, if any
ScriptVar10: publisher, if any
ScriptVar11: month and year
ScriptVar12: blank
ScriptVar13: mark for previous reference

plus ScriptVar50 as a general global variable.

The bibliographic information, by being spread out among 12 variables, allows flexibility of reference content (some citations will have e.g. paper titles but not city of publication) and the option of different data in the first and the subsequent reference. Thus, in the in-text/bibliography format, an in-text citation might be:

> Discussions of anthropological relativism make plain how easy it is to impute variable ends by failing to allow for the possibility that common goals are articulated differently in different circumstances [Kitcher 1992].

and the bibliography entry would be:

> Kitcher, Philip. "The Naturalists Return." *Philosophical Review*, **(101,1),** January 1992, p. 53-114.

done by using ScriptVar01 for the in-text citation, and ScriptVar02 plus 05 through 12 for the bibliography entry. Footnotes and endnotes work with the same idea but, since the first footnote

reference is the complete one, the first footnote would draw from ScriptVar02 and 05-12, with subsequent references built from ScriptVar01 plus something like "op.cit."

The main macro: Enter Reference

This macro, called by the AppleScript "Put Reference," is where most of the action is. In it, footnotes or endnotes are generated using the macro commands **New Footnote** or **New Endnote**, along with **Close Subdocument** to close the note window when the information has been entered. Enter Reference simply calls either new note command and then checks to see whether there's anything in ScriptVar13. If not (so the present reference is the first one in this paper), it types script variables 05-12 and closes the note window. If there is something in ScriptVar13, it types script variables 01-04. Something like:

```
New Footnote
If (ScriptVar13=0)
    Type Var (ScriptVar05)
    Type Var (ScriptVar06)
    Type Var (ScriptVar07)
    Type Var (ScriptVar08)
    Type Var (ScriptVar09)
    Type Var (ScriptVar10)
    Type Var (ScriptVar11)
    Type Var (ScriptVar12)
Else
    Type Var (ScriptVar01)
    Type Var (ScriptVar02)
    Type Var (ScriptVar03)
    Type Var (ScriptVar04)
End If
Close Subdocument
```

Two things to note: if a variable is empty, the Type Var command doesn't add anything, not even a space; and what we have so far doesn't give us formatting, e.g. book title in italics. We'll fix that, of course.

If, instead of a footnote or endnote, the user chooses the in-text/bibliography, the commands we'll use are actually those for a Table of Authorities, a legal reference format. This is excellent as well for the structure of standard academic citations, as shown in the examples in the boxes above.

These commands are **Mark Full Form** and **Mark Short Form**. As we'll use them, the macro will choose Full Form if ScriptVar13=0, indicating that this is a new reference. Otherwise, the macro uses Short Form. The identifiers for both forms are script variables 01-04. That data is all the Short Form gets; the Full Form command, though, opens a subdocument like the one for footnote entry, and puts script variables 05-12 there. When printed, that text doesn't appear at the bottom of the page or end of the file, as do footnotes or endnotes, but wherever you define a bibliography, using the same steps as to define a table of contents or an index.

The syntax for these commands is:

```
Mark Full Form (Section;Short Form)
Mark Short Form (Short Form)
```

where the Section is one of 16 (defaults to 1) and where "Short Form" is any identifying string, as is done with cross-referencing. For legal use in a Table of Authorities, a Short Form might be "Brown v. Board of Education," which reference would then appear everywhere that case was cited in the legal brief. The full reference would appear wherever defined and generated in the brief, just like an index. So the part of the Enter Reference macro with the code to enter this type of citation is:

```
If (ScriptVar13=0)
    Mark Full Form (1;ScriptVar01$ScriptVar02$ScriptVar03$ScriptVar04)
    ; code to enter ScriptVar05-12 here
    Close Subdocument
Else
    Mark Short Form (ScriptVar01$ScriptVar02$ScriptVar03$ScriptVar04)
End If
Right ()
End Macro
```

which finishes up the basics of this reference manager. Congrats if you've made it this far, since this is pretty complex design. Rest assured, the remainder: formatting the references, and how to tell whether the user wants a footnote, endnote, or in-text citation plus bibliography, is much easier.

Setting a citation format

The best way to work with a user preference is to put it in a global variable, and have the Enter Reference macro refer to it whenever the user wants to add a citation. Thus, in general terms:

```
If (GlobalVar01=1)
    Go (Footnote)
End If
If (GlobalVar01=2)
    Go (Endnote)
End If
; ========================================================
; In-text/bib code following, to run if GlobalVar01 hasn't sent execution
; elsewhere
; ========================================================
If (ScriptVar13=0) ; no X in the Mark field
    ; Mark Full Form etc.
End Macro
End If
; ========================================================
Label (Footnote)
; New Footnote etc.
End Macro
; ========================================================
Label (Endnote)
; New Endnote etc.
```

```
End Macro
```

and you could have some code in OnStartUp that posts a menu asking the user what he or she wants in the way of reference formats for the current session of WP – a pain, since the user probably wants to type a memo for the current session of WP. But if we let the user call the menu macro manually, it's just a little less elegant than I like to be. How can we get a value into a global variable automatically, when WP starts? If we can do this, the user's choice of formats will be remembered across work sessions, like the program preferences.

A data table

I set up a document called "Data" in the WP folder in System/Preferences, with a table in it, containing 12 rows. A macro could put a number in a given cell to represent a formatting option. For example, the number in column 2, row 1 could indicate the author-date or footnote or endnote choice. Take a look at this macro, called "Set Citation Format":

```
Menu (Var00;"Set Citation Format to:";{"Author-Date & Bibliography";"Endnote";"Footnote";"Endnote & Bibliography";"Footnote & Bibliography"})
Case (Var00;{1;Author-Date;2;Endnote;3;Footnote;4;EndnoteBib;5;FootnoteBib};cancel)
Label (cancel)
End Macro
;
Label (Author-Date)
Open Document (BootDir$"Preferences:WordPerfect:Data")
Position To Cell (1;2;1)
Select Word
Type (0)
Save
Close
Go (end)
;
Label (Endnote)
Open Document (BootDir$"Preferences:WordPerfect:Data")
Position To Cell (1;2;1)
Select Word
Type (1)
Save
Close
Go (end)
; and so on
```

which posts a menu asking users what kind of citation they want. If a user chooses Endnote, the macro opens the Data document – remember, the read-only variable BootDir is what your Mac has told WP is the System folder. Then we have the table navigation command Position to Cell, where the first parameter is the number of the table in the document, and the second and third parameters are the column and row in that table.

There may or may not already be a number in the cell we've chosen to use: **1;2;1** where the parameters identify the first table in the document (actually, the only table in this document), second column, first row. It doesn't matter, since the Select Word command will select that digit or empty space, and the selection will be replaced with whatever is typed. In this case, with the

choice of Endnote, the macro types a "1" and saves and closes the document. The user's preference is now on disk, ready for us to access with another macro.

That macro is "Get Citation Format," which might well be called by OnStartUp, to open the Data document and read the value of column 2, row 1, into a global which will be a reference for note entry in the present session. This code might look like:

```
Position To Cell (1;2;1)
Select Word
Copy
Assign (Var13;Clipboard)
```

and later we can assign Var13 to a global variable, to keep the data available for use.

Formatting instructions via the data table

While we're putting in and taking out the user's choice of citation type from a data table, let's define the formatting for each of the 12 transfer fields, to get figure 26. Why not just have the FileMaker field for, say, book title, be in italics and leave it at that? Two reasons: first, we want to have more flexibility in formatting than would be encouraged by putting that formatting in database fields. Your next editor might want book titles underlined, for example.

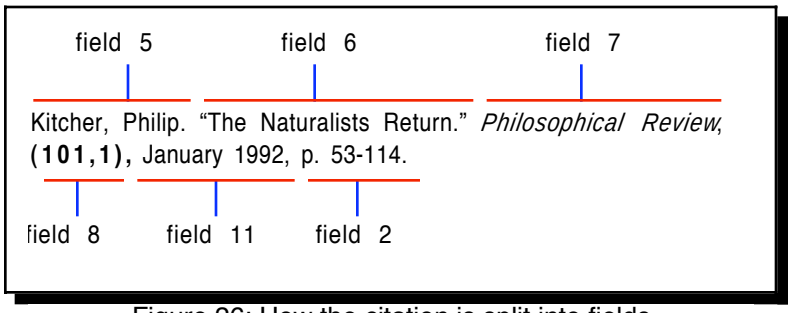
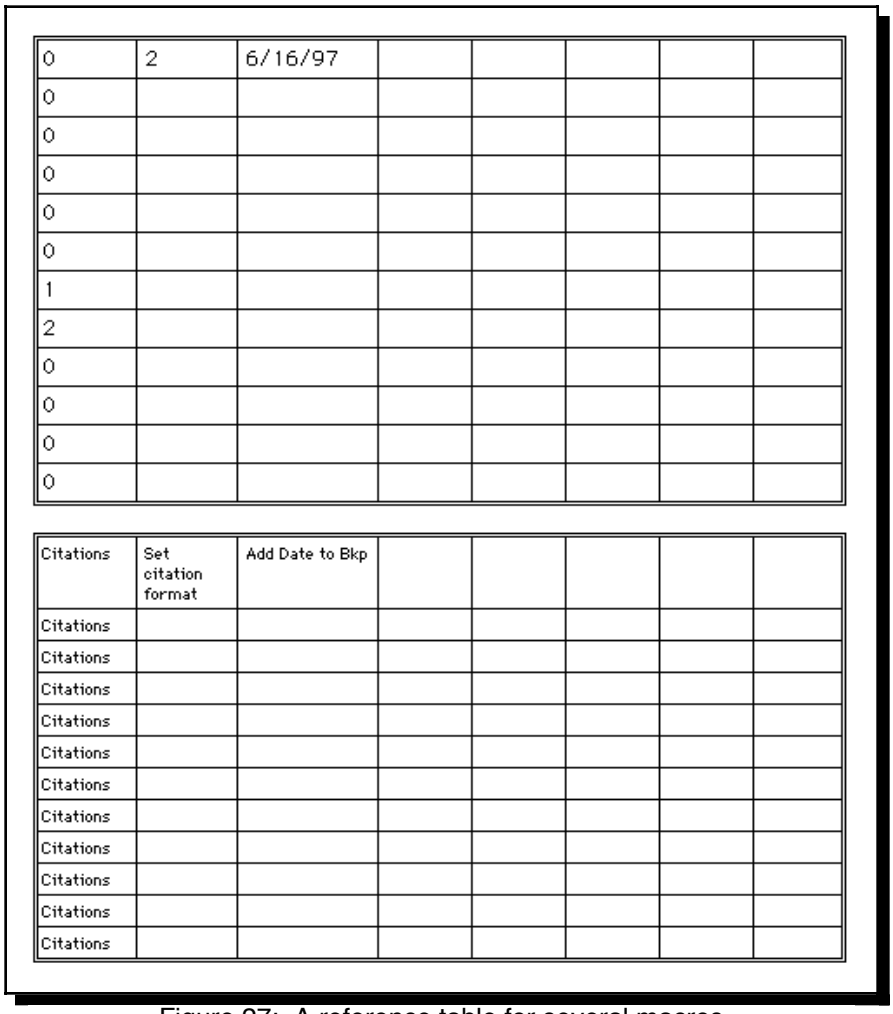


Figure 26: How the citation is split into fields

The second reason is that it would be more engineering to transfer formatting from FileMaker through AppleScript into WordPerfect variables. WP variables, for example, don't include style information – just text. So let's have the citation engine transfer text, and then format it while writing it to a reference. So each of the 12 fields needs a code to tell WP to write that field in the reference as plain text, or italics, bold, etc.

That's a simple code, so it can be a one-digit numeral. With a 12 row data table, I put the code for each field in succeeding rows in the first column. Our sample reference was then formatted with the data table shown in figure 27. The upper table contains the data; the lower tells which macro each cell serves:



0	2	6/16/97					
0							
0							
0							
0							
0							
1							
2							
0							
0							
0							
0							

Citations	Set citation format	Add Date to Bkp					
Citations							
Citations							
Citations							
Citations							
Citations							
Citations							
Citations							
Citations							
Citations							
Citations							

Figure 27: A reference table for several macros

where the zeros in most rows in column one tell WP to leave those fields as plain text. Row seven has the numeral 1, telling the macro we're about to write to make the contents of field seven italics. Row eight has the numeral 2, setting the contents of that field to bold. The numeral in column two, row one, sets citation type: author-date/footnotes/endnotes etc., as we saw earlier.

The macro that reads all these numerals is, again, Get Citation Format. It reads the data table with code segments like:

```
Position To Cell (1;1;1)
Select Word
Copy
Assign (Var01;Clipboard)
```

Twelve segments like this fill up variables 01 to 12 with the formatting data, and variable 13 with citation type. A line later in the macro:

```
Assign (ScriptVar50;
    Var01$Var02$Var03$Var04$Var05$Var06$Var07$Var08$Var09$Var10$
    Var11$Var12$Var13)
```

(this is all one line in the macro editor) writes all these local variables to ScriptVar50, which we're going to use just like a global variable – since it in fact is one, in addition to its special use with AppleScript.

So I've used script variables 1 through 13 to move data for an individual reference. If the user enters a reference, then runs another AppleScript/WP macro that overwrites the first 13 script variables, fine. The next time the user enters a reference, those script variables are again used for the citation data, and so on. The variable we don't want to overwrite is the one containing this formatting data. Since WP picks up data from AppleScript and writes them to script variables in numerical order, ScriptVar50 will be the last to be used for anything else.

To finish the tour, let's go back to the Enter Reference macro. This starts by picking apart ScriptVar50:

```
SubString (Var01;1;1;ScriptVar50)
SubString (Var02;2;1;ScriptVar50)
```

and so on, up to Var13. Var01-12 dictate the formatting for the 12 fields, and Var13 tells the macro which type of citation to write, as we saw above. If the user chose footnotes, the macro goes to this label:

```
Label (footnote)
New Footnote
If (ScriptVar13=0)
    Call (5-12)
Else
    Call (1-4)
End If
Close Subdocument
End Macro
```

which checks ScriptVar13 to see if this is a new reference. If so, it goes to a label I called "5-12" since it uses those script variables. This looks like:

```
Label (5-12)
Assign (Var49;Var05)
Call (format)
Type Var (ScriptVar05)
Assign (Var49;Var06)
Call (format)
; and so on
```

which puts the contents of (local) Var05 into Var49 and goes to a format subroutine, which looks like:

```
Label (format)
If (Var49="1")
    Attribute (On;Italics)
    Attribute (Off;Bold)
    Attribute (Off;Underline)
```

```
End If
If (Var49="2")
     Attribute (Off;Italics)
     Attribute (On;Bold)
     Attribute (Off;Underline)
End If
; and so on
```

so that if Var05 (formatting for field 5) contains the numeral 1, that goes to Var49 (the single variable for the subroutine), which then sets the italic attribute. The subroutine has enough of these segments so that italic, bold, underline or any combination is available.

If someone wants to use an attribute like shadow for an academic reference, they can do that themselves. :-)

After execution returns from the format subroutine, the macro types the contents of the script variable, and goes on to the next script variable.

And that's it! Since these macros are getting larger and more complicated as we go along, I've discussed them in pieces. It's important, though, that you look at the whole set of scripts, which you can download from the Info-Mac or Corel internet sites (see Appendix D for locations).

With AppleScripts as well as WP macros, it's very helpful to print out the code and draw arrows or make flowcharts, to and from the various labels, so you can make yourself a picture of how the whole thing works.

17: Advanced AppleScript

AppleScript has a fair amount of power, but much of it is invisible to the user who has only WordPerfect and AppleScript installed. Many more goodies are in the osaxen you can find, mostly shareware or freeware floating around the web.

Let's look at a particularly powerful osax, and design an engine around it that will use both AppleScripts and macros, and the one will call the other, possibly a couple of times. We'll make a search feature that dramatically enhances your ability to find information in your documents, by searching for *regular expressions.*

A regular expression, also called *GREP*, for Global Regular Expression Parser (a typically convoluted term reflecting the concept's UNIX origins) is a pattern in text. While WordPerfect lets you look for text itself, and formatting codes, it doesn't have this tool that technical writers especially find invaluable. For example, GREP will let you find the occurrence of any seven-digit telephone number followed by a word that begins with a vowel. You can look for any two capital letters followed by any single digit, followed by any two lower-case letters. Search for any word on a list, or one whose first two letters are within the first half of the alphabet, and last two letters are within the second half. Wildcard searches are exceptionally flexible.

The osax is "Regular Expressions," part of the excellent collection of free osaxen called "Script Tools," written by Mark Alldritt and Late Night Software. This is downloadable from http://www.scriptweb.com/scriptweb/, ftp://mirror.apple.com/mirrors/gaea.scriptweb.com/applescript/, or a number of other sites, including Info-Mac Mark also wrote Script Debugger, a much more advanced script editor than what ships with AppleScript.

When you find an osax that looks interesting, the first step is to look at that file's AppleScript dictionary, also called the *AETE*, for Apple Event Terminology. You can access the dictionary of any osax, or any application, by dragging its icon over the Script Editor, or you can open the dictionary from Script Editor's file menu.

How to read a dictionary

The dictionary for Regular Expressions appears in part as shown in figure 28, after you click on the subheads in the column on the left. Those subheads show the AppleScript *commands*, in regular type, and *objects*, in italics. Commands are just the things you can do, and objects are things you can do them with. We'll only use the first two commands, and the first two objects of the class (or type, as it were) "Match Reply."

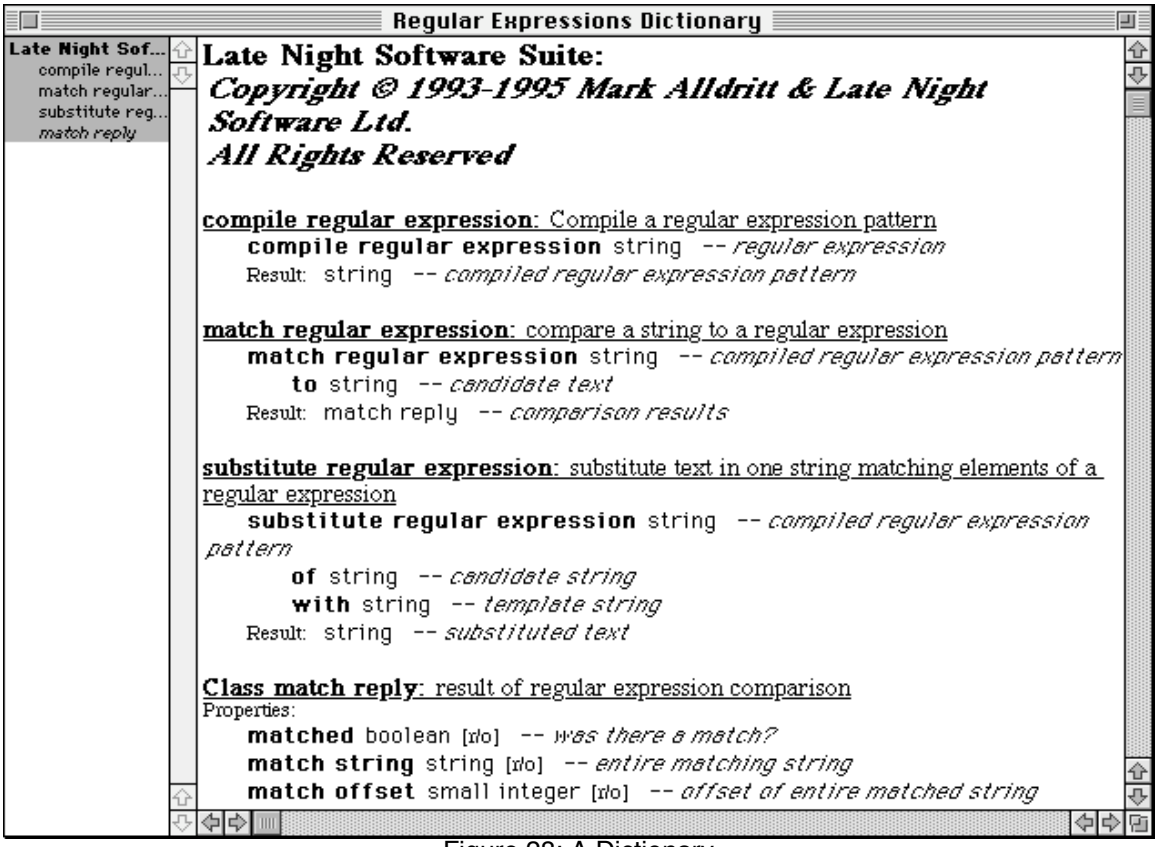


Figure 28: A Dictionary

The commands start with **compile regular expression**, detailed on the first line. The underlined bold text says what it is, and the following underlined plain text says what it does. The bold text on the next line is the exact syntax of the command, followed in plain text with the kind of parameter the command takes. The next line shows the result: what happens when you run this command.

So we'll use the first command to compile a regular expression from the user's input, and the second command to match the regular expression to the candidate text – whatever the user's searching in. After running those commands, we'll work with the Match Reply properties of **matched**, which is a boolean (true or false), and **match string**, which is the text found (if matched is true) that matches the pattern we gave it.

For example, GREP uses a plus sign to stand for one or more of the preceding character. Input a pattern of "bus+" and our program will look at the candidate text for anything matching that pattern. If it finds anything, matched will then be true, and match string will be the text it found: in this case, "bus" or "buss" or "bu" and any number of "s" characters. The match string will be the longest one matching the pattern.

Since this osax returns the first match it finds, it would be of limited value to have it look at all the text in a document. Having it look within a selection is a useful alternative.

Starting at the beginning

Let's start with a dialog asking the user for a pattern to match. In the most straightforward way, that would go:

```
display dialog "Enter Pattern" default answer "" buttons {"Cancel","Match"} default button
2
```

and what the user enters then lands in a reserved (i.e. read-only) AppleScript variable called, appropriately enough, **result**, and we could work with that. But the contents of result are replaced as soon as the script does something else that has a result. We can give our result some longevity by putting its value into another variable we'd name, as in:

```
set dialogResult to the result
```

but AppleScript lets us combine these two lines into:

```
set dialogResult to display dialog "Enter Pattern:" default answer "" buttons {"Cancel","
Match"}
```

and you can use parentheses to make the two parts of the line clearer, as in:

```
set dialogResult to (display dialog . . . )
```

So dialogResult is a variable containing the user's input. That input consists of a button choice and some text, so we can define the variable **pattern** as the text entered when the button Match is clicked (if the user clicks Cancel, the script just stops. We'll change that in a bit). Finally, theText is a variable containing text we'll be searching in.

All this looks like:

```
set pattern to text returned of dialogResult
set theText to contents of selection of window 1 of application "WordPerfect"
set searchPattern to (compile regular expression pattern)
set matchResult to (match regular expression searchPattern to theText)
```

Where **matchResult** will have the properties shown in the dictionary for class **Match Reply**. Of those, **matched** and **match string** are what we want to work with. We could do something like:

```
if matched of matchResult then
    set matchString to match string of matchResult
else
    display dialog "I found nothing matching “" & pattern & ¬
    "”." buttons "OK" default button 1
end if
```

to give us the string of text matching the pattern, in the variable matchString. And that's the heart of the matter.

And then my problems began . . .

What we have now will look in selected text in the active WP document, match a pattern to a string in that text, and give us that string. What shall we do with that? We could put matchString in a dialog, but that's not much help. Better to select the match. According to the dictionary (below what's shown in figure 28), we could get the **match offset** – the number of characters from the start of the file to the match, but I couldn't get that to work very well in WP. It would get close but, depending on formatting and other factors, wouldn't accurately select the match string.

This turns out to have a simple solution, though: as long as we have the string in a variable, we can then have a WP macro find it. I wrote a macro called GREP that's basically just a Find command. Remember that an AppleScript can call a WP macro and send along parameters which WP gets in the form of script variables. So we'd have an AppleScript:

```
if matched of matchResult then
    set matchString to match string of matchResult
    tell application "WordPerfect"
        activate
        Do Script {"GREP", matchString}
    end tell
end if
```

calling this macro. It just does a find, with matchString, now in ScriptVar01, as the find string:

```
Find/Change Direction (Within Selection;Wrap Around)
Find/Change Where ({Current Doc})
Find/Change Match (Partial Word;Case Sensitive;Alphabet Insensitive;CharRep Insensitive;{Text Only})
Find/Change Action (Select Match)
Find String (ScriptVar01)
Find
```

and we've done what we set out to do: get a pattern from the user, match it to candidate text and, if a match was found, select the string within the text. The rest is just cleanup: user support, error handling, and polish. Let's start with user support.

Help!

The symbols that describe a pattern, using what's called *metacharacters,* become automatic with enough use but, until the user reaches that point, he or she would find an on-line help reference very useful. That's easy enough to do, with another display dialog like:

```
display dialog "Help" & return & "_____" & return & return & ¬
".     Any char
\\    Precedes literal
|     Logical OR

+     One or more of preceding
*     Zero or more of preceding
?     Zero or one of preceding

[ ]   Encloses any match
```

```
[^]  Excludes enclosure
( )   Sets operator precedence" buttons "OK" default button 1
```

and I've shown two ways to put multiple carriage returns in a dialog. The first is just to say "return" within the display dialog line, and the second is to physically include those returns in the text.

Another point to note is that a backslash represents a literal character in GREP syntax *and* in AppleScript syntax: to get a backslash into the dialog, you have to type two of them into the script.

Flow control in AppleScript

Now that we have the body of the script and a help dialog, how do we fit them together? There's a little difference here between flow structure in AppleScript and in WP macros. In WP, we might give the user a choice on a menu; if the user chose "Help," we could have the line

```
Call (Help)
;
Label (Help)
; alert with help
Return
```

and execution would return to the line below the Call command. The other way to direct execution is to use a handler. You learned the error or cancel handler on page 62, as in:

```
On Error (errorRoutine)
```

and AppleScript lets us name handlers anything we want, and uses them like WP's cases and labels. So we could go from the initial dialog to the help window with:

```
set dialogResult to display dialog ¬
    "Enter pattern: " default answer pattern ¬
        buttons {"Help", " Cancel ", "Match"} default button 3
set userChoice to button returned of dialogResult
```

and, if userChoice is "Help" then we send execution there with:

```
if userChoice is "Help" then
    help()
end if
```

Where "help()" is a label for a subroutine. The parentheses are required; here they are empty but could just as easily be filled with a parameter to use in the subroutine if needed. None needed here; the display dialog command just produces figure 29:

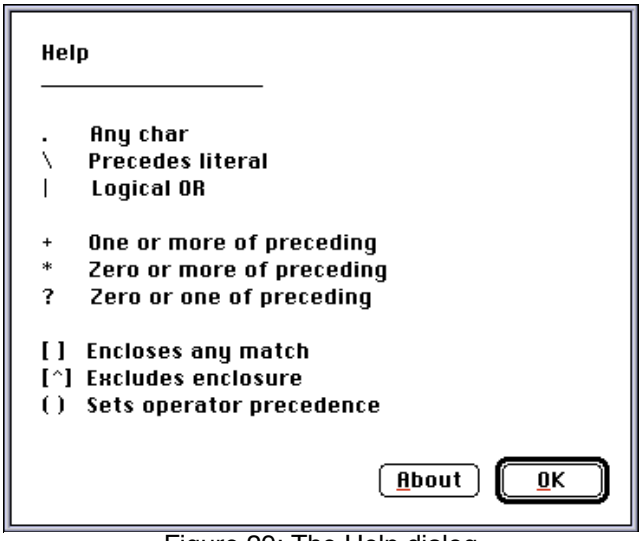


Figure 29: The Help dialog

OK, that gets us to the help routine. How do we get back, assuming that after the user peruses the help screen, he or she will want to go back to the initial dialog?

We can think of the entire script as being within a handler called **Run.** A script can then be seen as having the structure:

```
on run {}
    -- code here
end run
```

Explicitly declaring a run handler is most useful when a script is designed as a droplet (drag and drop an icon over it to get something to happen) as well as an applet (double-click it). In that case, the script's structure would include:

```
on open {}
    -- code to execute with drag and drop
end open

on run {}
    -- code to execute if double-clicked
end run
```

and, since we aren't building a droplet in this case, the Run handler can be assumed. We can also call the body of the script "top level commands" since they aren't contained in a specific handler. So, to get from the Help handler back to Run (or the top level), we just add a line to say so:

```
if userChoice is "Help" then
    help()
end if

on help()
    -- text of dialog
run
end help
```

so, if users click the Help button, they get the help screen. As soon as they click OK, they go back to the top of the run handler – in this case, the start of the top level of the script – the initial dialog. With the Help button in it, there's only a couple more things to do to make that dialog look spiffy, as in figure 30:

Figure 30: The initial GREP dialog

the first of which is to get that icon. That's part of the system file, as you see when perusing a copy of the system with ResEdit. There are a few interesting icons, and all you need is a number, as in:

```
display dialog "Hello" buttons "OK" default button 1 with icon -16396
```

For more ordinary uses, icon 0 is stop, icon 1 is note, and icon 2 is caution.

The only other design point for this or any dialog is to space things out so they look good. Spaces as well as returns work here; note, though, that a dialog can't have more than 255 characters overall.

We now have a pretty good working script, to find patterns within a selection, and to offer a help screen for the user's convenience. What now? Let's get fancy. It would be nice if the script were able to remember search patterns from one use to the next. We can do this with another feature of AppleScript, called a *script property.*

Script properties

Think of a script property as a persistent variable – one that the script saves with itself on disk, so it remembers the property's value from when you last ran the script. They're easy to set up, and you name them and use them just like variables. Somewhere towards the top of your script, before the action begins, add this line. It *declares* a property:

```
property pattern : ""
```

so that when you save the script in its editor, the property "pattern" is a null string. When you run the script, and pattern is given a value, that value is saved with the script – it persists, and will be there the next time you run it. Then, instead of telling display dialog that the default entry is "" (i.e. always blank), you can tell it:

```
set dialogResult to display dialog ¬
"      John's WordPerfect GREP" & return & return & ¬
```

```
"Enter pattern: " default answer pattern ¬
buttons {"Help", " Cancel ", "Match"} default button 3 with icon -16396
```

and the default entry will be blank the first time the script is run. But since pattern is a property, the next time the script is run, the text entry area in the initial dialog will be what was searched for last time. Users appreciate this.

Text item delimiters

As an aside, a script property provides another way to pass data to WP. Instead of specifying each parameter, you can specify the **delimiter** separating items on the clipboard, say, or in a selection. Then tell WP to do a script taking the items of the list as parameters. Thus, a later version of Citations just has AppleScript tell FileMaker to put fields of a record – each separated by the diamond symbol "◊" – onto the clipboard. WP then gets the delimited data, as in:

```
property text item delimiters : ASCII character 215

tell application "FileMaker"
    activate
    do script "Citation"
    set theClip to the clipboard as string
end tell

tell application " WordPerfect"
    activate
    Do Script {"Citation"} & text items of theClip
end tell
```

Up to 50 text items can be handled this way, and the delimiter can be whatever's convenient: a hard return, for example. The clipboard access is provided by Jon's Commands, a free osax that everyone ought to have. Now – back to our project

Global variables

You set these up like script properties, but they hold a value only while the script runs. Like WP but, again, you can name any number of them with any name that isn't reserved by AppleScript, like "result" or "copy". Local variables (those not declared as global) hold their value only within the handler in which the value is declared. So our program will have:

```
global theText
```

right at the top, along with the property declaration.

The Cancel business

We have a Cancel button in the initial dialog, but anything that returns "Cancel" just ends the script. What if you want to do something else? Add spaces to the string in the button, and make a handler for it.

```
set dialogResult to display dialog ¬
    "      John's WordPerfect GREP" & return & return & ¬
    "Enter pattern: " default answer pattern ¬
    buttons {"Help", " Cancel ", "Match"} default button 3 with icon -16396
set userChoice to button returned of dialogResult
```

where I put spaces on both sides of "Cancel" so the button would look even in the dialog. Then we do:

```
else if (userChoice is " Cancel ") then
    tell application "WordPerfect" to Do Script "GREP"
```

so WP will run its macro (no need to tell the application to activate first). If WP gets this Do Script command without a parameter (i.e. ScriptVar01 is not given a value, then I have the WP macro just redraw the screen and quit. That looks like:

```
Display (Off)
If (!ScriptVar01)
    Go (end)
End If
```

and I turn display on (redraw WP's screen) at the end, since AppleScript sometimes leaves WP a little messy. If there is a value in ScriptVar01, and I use that to find a match, I afterwards assign that variable a value of 0, because ScriptVar01 is a global in WP. So if I left the match string in, the command would find that again the next time the user clicks Cancel. Not too smooth.

Error handling

As always, this is a big part of a good script. AppleScript does this with a try statement, which ends with an error handler. As in:

```
try
    -- some code that might work
on error
    -- what happens if it doesn't work
end try
```

What you often want to happen when something doesn't work is just post a dialog and tell the user.

So we start things off by looking for some text the user has selected to search in. AppleScript syntax is "contents of selection", since just saying "selection" returns its properties, e.g. number of words in it. Let's put those contents in a variable named theText. If there ain't no contents, though, we get an error. So tell the user what's wrong, and we have:

```
try
    set theText to contents of selection of window 1 of application "WordPerfect"
on error
    tell application "WordPerfect"
        activate
        display dialog "Please open a WordPerfect " & ¬
            "document and select text in it before running WP GREP." buttons ¬
            "OK" default button 1 with icon 2
```

```
        end tell
    return
    end try
```

and the return command in the next to last line returns execution to the handler that called this subroutine. Since we're in an implied run handler, it just ends the script.

Rather than a static error dialog, though, we can get a number and message, specific to that error, to the user, if the app or osax returns one to the script. That syntax looks like the next try statement in the script:

```
try
    set searchPattern to compile regular expression pattern
    set matchResult to match regular expression searchPattern to theText
on error errormsg number errornum
    activate me
    display dialog "Error " & errornum & ¬
        ": " & errormsg buttons "OK" default button 1 with icon 2
    set pattern to ""
    return
end try
```

where the parameters to the on error handler declare the variables for the error message and number. The dialog picks those up, and we're in business.

Timeouts

An AppleScript command times out in a minute unless you have something to say about it. I wanted to give the user, especially one new to GREP, more time, so I added timeout statements to a couple of handlers. Plus this and that, and the AppleScript and WP macro ended up as in the next section.

The scripts

Applescript:

```
--osax: Regular Expressions from Script Tools 1.3.6 by Mark Alldritt

property pattern : ""
global theText

with timeout of 3600 seconds
    try
        set theText to contents of selection of window 1 of application " WordPerfect"
    on error
        tell application " WordPerfect"
            activate
            display dialog "Please open a WordPerfect " & ¬
                "document and select text in it before running WP GREP." buttons ¬
                "OK" default button 1 with icon 2
        end tell
        return
```

```
        end try
        if theText = "" then
            tell application " WordPerfect"
                activate
                display dialog "Please select text first." buttons "OK" default button 1 with
icon 2
            end tell
            return
        else
            tell application " WordPerfect"
                activate
                set dialogResult to display dialog ¬
                    "      John's WordPerfect GREP" & return & return & ¬
                    "Enter pattern: " default answer pattern ¬
                    buttons {"Help", " Cancel ", "Match"} default button 3 with icon -16396
                set userChoice to button returned of dialogResult
                set pattern to text returned of dialogResult
            end tell
        end if
    end timeout

    if userChoice is "Help" then
        help()
    else if (userChoice is " Cancel ") then
        tell application " WordPerfect" to Do Script "GREP"
    else if (pattern is "") then
        tell application " WordPerfect"
            activate
            display dialog "No pattern entered." buttons "OK" default button 1 with icon 2
        end tell
        run
    else if userChoice is "Match" then
        OK(pattern)
    end if

    on help()
        with timeout of 3600 seconds
            tell application " WordPerfect"
                activate
                display dialog "Help" & return & "_____" & return & return & ¬
                    ".      Any char
\\      Precedes literal
|       Logical OR

+       One or more of preceding
*       Zero or more of preceding
?       Zero or one of preceding

[ ]     Encloses any match
[^]     Excludes enclosure
( )     Sets operator precedence" buttons {"About", "OK"} default button 2
            end tell
        end timeout
        if button returned of result is "About" then
            credit()
        else
```

```
            run
        end if
    end help

    on credit()
        with timeout of 3600 seconds
            tell application " WordPerfect"
                activate
                display dialog ¬
                    "                John's WordPerfect GREP


    This software is free for non-commercial distribution and use." & ¬
                    "Comments and suggestions welcome at jcr2@cornell.edu


    by John Rethorst
    ©1997, 1998.
    All rights reserved." buttons "OK" default button 1
            end tell
        end timeout
        help()
    end credit

    on OK(pattern)
        try
            set searchPattern to compile regular expression pattern
            set matchResult to match regular expression searchPattern to theText
        on error errormsg number errornum
            activate me
            display dialog "Error " & errornum & ¬
                ": " & return & return & errormsg buttons "OK" default button 1 with icon 2
            set pattern to ""
            return
        end try

        try
            if matched of matchResult then
                set matchString to match string of matchResult
                tell application " WordPerfect"
                    activate
                    Do Script {"GREP", matchString}
                end tell
            else
                tell application " WordPerfect"
                    activate
                    display dialog "I found nothing matching:" & return & return & pattern ¬
                        buttons "OK" default button 1 with icon 2
                    Do Script "GREP"
                end tell
            end if
        on error errormsg number errornum
            activate me
            display dialog "Error: " & errornum & return & return & errormsg buttons ¬
                "OK" default button 1 with icon 2
        end try
```

```
end OK
```

GREP macro:

```
Display (Off)
If (!ScriptVar01)
    Go (end)
End If
;
Find/Change Direction (Within Selection;No Wrap)
Find/Change Where ({Current Doc})
Find/Change Match (Partial Word;Case Sensitive;Alphabet Insensitive;CharRep Insensitive;{Text Only})
Find String (ScriptVar01)
Find
Find/Change Reset
Assign (ScriptVar01;0)
Label (end)
Display (On)
```

The practice you get from dissecting other people's work is just as important in AppleScript as in WP – maybe more so, since by making the language easy to use, Apple engineers made the syntax a little vague. Things that dictionaries say should work don't quite, and looking at solutions in practice is time well spent.

18: Automating Data Entry

We'll be looking at ever more exotic and powerful macro commands and structures as we go along, but I'd like to include not only more commands, but more applications for those commands: case studies, real-world solutions that might be what you're looking for or, more importantly, that can give you the inspiration and the means to create, on your own, what you're looking for.

I received interesting email from Dr. T., a dermatologist at Johns Hopkins University Hospital. He described the time-consuming method a physician uses to enter symptom description, diagnosis and treatment plans on a patient's chart, writing the information longhand. Additionally, gathering quantitative data is an expensive part of research, manually transcribing and tabulating each field from each chart onto worksheets for statistical analysis. A medical software supplier had offered a solution in the form of a FileMaker template – for $3000. Was there a better way?

Certainly. Let's design one.

What the physician wants to do is go to a Mac in the examination room and open a template for the patient's condition. She would then click on buttons that opened short dialogs for text and number entry – for the patient's name and age – and buttons that post menus where options, such as gender, localization of the dermatological affliction, and treatment, are limited in number. By clicking on buttons, making menu choices and, where necessary, entering text, the physician can complete a computer-generated chart for the patient quickly and accurately. By virtue of its structure, because it's on disk, and because terminology is standardized, researchers can later pick data out of several hundred charts at a time.

The 'buttons' I'll use will be WP 3.5's hyperlinks, where a word can be linked to a bookmark, to a URL on the Web, or to a macro. Our links will run macros. If you're using WP 3.0x or 3.1, you can approximate this effect using buttons on the Button Bar that link to the same macros.

As a dermatologist, Dr. T. treats many patients for acne. As the attending physician, he would examine a patient, then turn to a Mac running WordPerfect and, from the Template submenu on the File menu, choose a template we might simply call "Acne." A new, untitled document then opens with the contents of that template.

The first paragraph would read like the following. The underlined words are links, so they appear in blue on screen:

> Name is a age year old race gender who consults for the presence of lesions localized primarily on the localization which have been present for approximately time, and treated in the past with treatment.

Define a link by selecting the word or words, and choose Create Hyperlink from the BookMark menu. Link to a macro in the resulting dialog box, shown in figure 31.

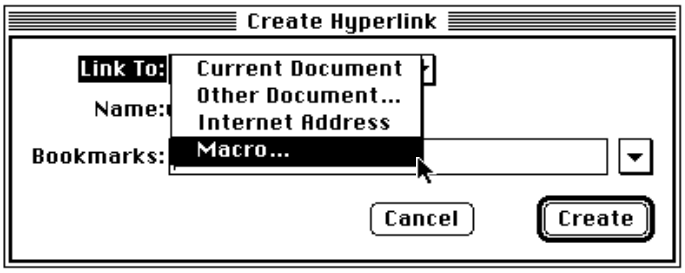


Figure 31: The Create Hyperlink dialog

Oops, we haven't written the macros yet. OK, let's do that.

"Name" will of course need a dialog to enter text, so we'd have something like:

```
Get Text (Var01;"Patient Name";"Enter patient's name:")
```

followed by some code to replace the link with the name:

```
Word Left (Select)
Type Var (Var01)
```

How did I know that "Word Left (Select)" would select the linked word and link codes, so that typing the variable would replace them? I didn't, so I fiddled around and tested things.

Entry for age could be done with Get Text as well, since the number is, in this case, essentially text. Get Integer is a another choice, though, since the command lets you check against limits, with the syntax:

```
Get Integer (Variable;Lower Limit;Upper Limit;Title;Prompt)
```

so that if the user enters an age greater than, say, 125, the macro will post an alert advising that the number is too high. For this use, though, I'd be more inclined to put the command's lower limit at 0 and the upper limit at 32767 (the highest number a variable can hold) and check for accuracy with code that will ask about a high number, but allow it:

```
Label (age)
Get Integer (Var02;0;32767;"Age";"Enter Patient's Age:")
If (Var02>125)
    Confirm (Var03;Caution;YesNoCancel;Var02$" seems high. Are you sure?")
    If (!Var03)
        Go (age)
    End If
End If
```

to get figure 32, if the age entered were 126:

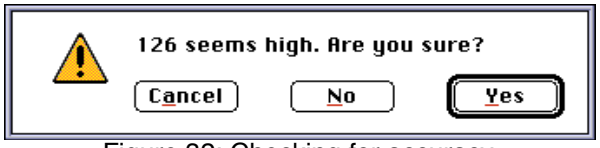


Figure 32: Checking for accuracy

and I've tossed a couple of shortcuts in there. Remember when we learned the Confirm command, we followed it with a Case command, leading to different labels. We could have that here, with:

Case (Var03;{1;Yes;0;No}**)**

but why bother? If the user clicks Yes, we want macro execution to proceed to the next step. If Cancel, we want the macro to stop, which the Confirm command does anyway (unless preceded by an On Cancel handler). Only if the user clicks No, thus giving Var03 a value of 0, do we want to repeat the Age entry. So we can use a simple If statement, looking for that value of 0. That could be:

If (Var03=0**)**

but the exclamation point – the not operator – is shorthand for the same thing. You can use this syntax with any variable, so that:

If (!DocumentModifyFlag)

is true if the document does not have unsaved changes.

Another, small but elegant, touch: the prompt "Enter Patient's Age," is entered in straight quotes – required in the macro editor – but the apostrophe within the prompt is curly, entered with Option-Right Bracket on the Dvorak keyboard, or Shift-Option-Right Bracket on Qwerty. Dialogs look better with curly apostrophes or quotes, as does anything else.

For quotes which are part of the command syntax (a **delimiter**) the macro editor lets you enter either single or double quote marks, so that use of the other within the piece of text doesn't confuse things. So:

Alert ('Say "Sneeze."'**)**

uses single quotes to define the parameter or argument – the alphanumeric expression for the entire alert, with double quotes within the text. Useful if, for some reason, you want straight quotes to be part of the dialog.

For further complexity you can use the **ordinary character delimiter**, a backslash (\). This tells WordPerfect that the character following it is a regular character, as in:

Type (This is a very good (or a very bad\) thing.**)**

so the closing parenthesis in the text is not taken to be the end of the command parameter.

Back to business. The largest part of data entry in the chart won't be typed names and numbers but menu choices, for all data where options are limited. The next links, race and gender, thus go to menus, as does localization:

```
Menu (Var06;"Localization";{"Face";"Face and Back";"Back";"Chest"})
Word Left (Select)
Case (Var06;{1;face;2;faceBack;3;back;4;chest};cancel)
Label (cancel)
End Macro
Label (face)
Type (face)
End Macro
Label (faceBack)
Type (face and back)
End Macro
Label (back)
Type (back)
End Macro
Label (chest)
Type (chest)
```

to put this data into the chart.

The **Type** command is limited to 255 characters. If the text you want to enter has more than that, just use the Type command again.

More complexity: menus calling menus

For much of the medical data, a simple menu would either not offer enough choices, or be too long and unwieldy. The command length in the macro editor is 512 bytes, so a menu with numerous choices might run out of room, as well as being a nuisance to work with. So we can have a macro with menus going to labels which run other macros, using the **Run** command:

```
Run ("Retin A")
```

The "Treatment" menu could then appear as in figure 33:

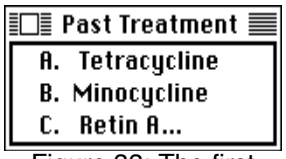


Figure 33: The first treatment menu

where option C will produce the menu shown in figure 34:

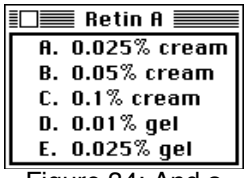


Figure 34: And a subsequent menu

and each choice in figure 34 would type "Retin A" plus the data shown. Note that in figure 33, "Retin A" has an ellipsis after it. This is a Mac standard interface guideline to tell the user that she has more decisions to make. With enough menus branching to menus, any degree of complexity in data entry can be achieved.

What we have at this point is an entry system that replaces the links in the sample paragraph with data of the appropriate type. While the resulting text paragraph would be optimal for some uses, I'd now like to look at the advantages of a table over paragraphs of text – especially for later data collection.

Using tables

Tables are wonderful things. They're great for flexibility in formatting and page layout, even in places where you'd never think of using a table. If you set the borders not to print, your readers won't ever know that you've used a table, thinking instead that you did the formatting in some more laborious way.

Another advantage to tables is specific to macros: your macro can locate a particular piece of data in a table much more easily than anywhere else. We saw this earlier with reference formatting data, but it will be more valuable, even critical in the later step of quantitative data collection such as we'll look at now, where a macro will open 500 charts automatically, get e.g. the age and treatment for each patient, and list this data in another file, ready for statistical analysis and graphing.

To work within a table, we'll use the Position to Cell command, as in:

```
Position to Cell (TableID;Column;Row)
```

where TableID is a read-only variable containing the number of that table in the current document. To specify the current table, use the variable by name in the command, such as:

Position To Cell (TableID;2;3**)**

to go to the second column, third row in the table containing the insertion point.

What we would want to do is then put the links in the left column and drop the data in the right column, as in figure 35:

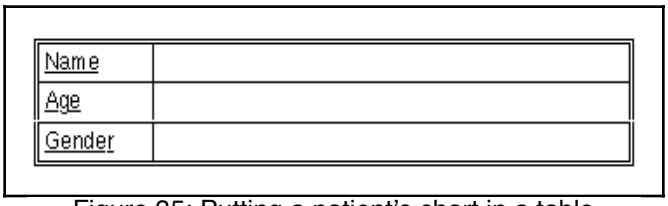


Figure 35: Putting a patient's chart in a table

and here I ran into a problem. With the current program version, I was unable to create a link in a table cell: everything following in the table became part of the link. Not to worry – I found that I could create the link elsewhere, cut and paste it into the table cell, and everything's fine.

Selecting the linked word to cut and paste required using the codes window, though. Simply double-clicking on the linked word selected the text but not the link codes. So I opened the codes window and used Shift-Right Arrow or Shift-Left Arrow (you can also Shift-Click), to get figure 36:

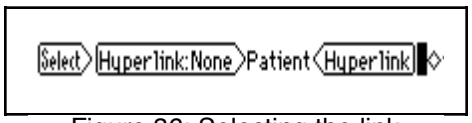


Figure 36: Selecting the link codes

at which point I chose Cut.

In order to work in the table, the "Name" macro would lose the line "Word Left (Select)" and get a Position to Cell line specifying "TableID;2; 1" and we're set. A general caution when using Position to Cell: make sure there isn't text selected or (with the present program version) the command won't work.

Another advantage of dropping the data in a table cell arises with data such as past treatment, where a patient might have been treated with both Tetracycline and Retin A. Rather than having exhaustive menu choices for multiple treatments, the physician could click the Treatment link once and enter the first drug, then click a second time. The table cells expand to contain all the data.

Getting the data back

The subsequent macros, to retrieve data from many charts and list it for analysis, would use a method similar to the one I discussed in chapter 14, to globally find and change text in all files in a folder. This time, the macro would open each file in turn, position to cell whatever, copy, cycle windows to your data collection document, find the next cell in a table there, and paste. Finding the next cell is easy: increment a variable and use that for the table row, as:

```
Assign (GlobalVar01;GlobalVar01+1)
Position To Cell (TableID;2;GlobalVar01)
```

using a global so you could run the macro several times in one session and not lose count of where you were in your data collection table.

Document variables

Another step in convenience here is, instead of using a global variable, using a **document variable.** These are just like other variables, but are document-specific, and are saved with the document. There are ten of them, **DocVar0** through **DocVar9**. You could then keep track of what cells in the data collection table had been filled, automatically, across multiple sessions of WP.

As a caution, when I write to a document variable, I make some other minor change in the document as well – adding a space at the end, say – since, depending on the program version, changing only a document variable might not trip the document modify flag, and the variable's new value then won't be saved.

I use the same caution when editing a macro installed in a document.

Putting menu choices up front

At this point we have what Humphrey Bogart might have called the beginning of a beautiful data entry system. Let's add one modification which will serve two purposes: make it much easier for a non-macro-literate associate to revise the system for another medical condition, and solve the problem that might arise if a menu gets too lengthy. Remember, the maximum command length in WP macros is 512 bytes, and a menu can easily get longer than that.

This modification is putting values in local variables first, followed by menus and commands that address those variables. For example, a menu like:

```
Menu (Var49;"Choose a Snack";{"Apple";"Banana";"Coconut"})
```

could be done as well with:

```
Assign (Var00;"Choose a Snack")
Assign (Var01;"Apple")
Assign (Var02;"Banana")
Assign (Var03;"Coconut")
Assign (Var04;"")
;
Menu (Var49;Var01;{Var02;Var03;Var04;Var05})
```

so that the menu command itself won't get too long. I added an empty variable, Var05, to show that you don't need to fill up all the variables: run this macro, and you'll see that only three choices appear on the menu. The meta-template macros could use all 50 local variables, and your associate could drop text into as many of them as necessary. She wouldn't need to dig into the script, since variable assignment will be right at the top. If you remember how daunting macro code might have looked to you when you started, you know how appreciative your associate will be.

And there we go! I think this chapter might be especially useful, as you can write these menus and text entry commands for a wide range of data entry purposes, use edits to check accuracy, achieve a consistency of terminology that's most helpful, and make data manipulation an order of magnitude easier. My thanks to Dr. T. for this great idea.

19: Going Around in Circles

Macro systems we've looked at up to now have been fairly linear in the respect that they let the user make a choice from, say, a menu, and get an action. They then go to sleep until the user calls them again. The macros haven't done much to keep track of things while you weren't running them.

Time for that to change. Let's look at a macro set that gives the user better control over multiple open documents. You may have four or five files open, comparing different drafts, opposing arguments or whatever. You might like a sort of bookmark feature to go back and forth to specific places among these files, but bookmarks themselves take longer to set and, in this case, outlast their usefulness. A bigger drawback, though, is that they're document-specific (either the program bookmarks in version 3.5, or my bookmark macros for 3.0 and 3.1).

This alternative, "John's WP Previous Positions," can be set in a second, and works across all open documents. The menu looks like figure 37:

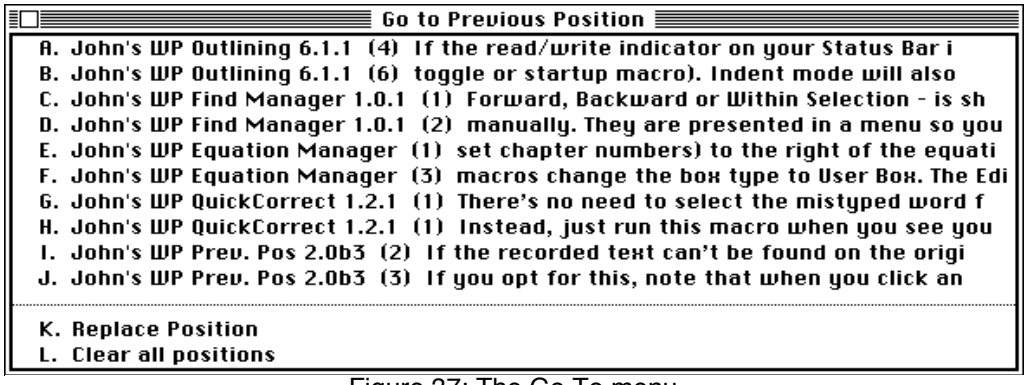


Figure 37: The Go To menu

where each option has the document name; then, in parenthesis, the page number, and then several words from the position. The menu is as wide as your screen, so enough of the text can be displayed to be recognizable.

When you click on an option, or type the letter to its left, the macro brings that document to the front, goes to the specified page, and searches for the text, selecting it. If editing has moved the text so it doesn't appear on the specified page, the macro posts the dialog shown in figure 38:

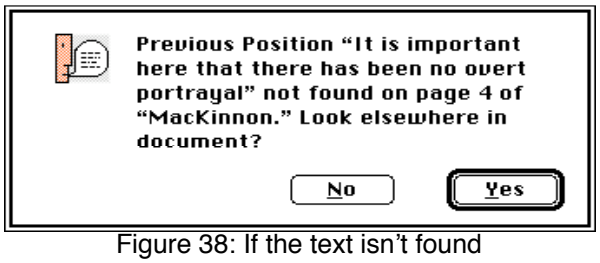


Figure 38: If the text isn't found

If you click Yes, the macro searches the whole document. If it can't find the document because you closed it, the macro tells you.

As part of the basic design, the macros remember the ten most recently assigned positions. When all ten memories are filled and you assign another position, the oldest one is replaced. We'll see how to do that, with a 'carousel' design you can use for any number of things.

Vary that variable

This set consists of two macros: "Assign Previous Position" and "Go to Previous Position." With the Assign macro, we start off with a global variable, which we increment by 1, every time the macro runs:

```
Assign (GlobalVar12;GlobalVar12+1)
If (GlobalVar12>10)
    Assign (GlobalVar12;1)
End If
```

so that GlobalVar12 will always have a value between 1 and 10. That value determines which of the 10 menu options is replaced by the most recently assigned position. We call this sort of thing a **counter**, and you can find a lot of uses for it.

With that variable in place, we can then set up a case command:

```
Case (GlobalVar12;{1;one;2;two;3;three;4;four;5;five;6;six;7;seven;8;eight;9;nine;10;
ten})
```

so that execution goes to labels designated by the spelled-out numbers, based on the numeric value of GlobalVar12. Why did I spell out the numbers for labels? Just to keep things straight. You can use names of cars for labels if you want – just so the case command and the labels match.

A label looks like:

```
Label (one)
Call (select)
Assign (GlobalVar13;DocumentName$"  ("$LogicalPage$")  "$Clipboard)
Go (end)
```

so global variable 13 will make up one of the choices on the menu.

At a later point we'll be picking apart these globals to get the document name, page, and text that was on the clipboard, and we'll need a way to distinguish those three pieces of data. The parentheses wouldn't do very well by themselves, since someone might easily have parentheses in their document name, so I put two spaces on either side of the parentheses.

Note that execution branches right off to a Select label which, as you might expect, selects text to designate the position. We'll get to that in a minute. Since we go to Select with a **Call** command, execution will return to the line just below the Call command. That next line assigns three things to the next global variable: the document name, the logical page, and the contents of the clipboard (or as much as will fit in the 255 character variable limit).

Ten globals – GlobalVar13 through GlobalVar22 – make up the menu posted by the Go To macro.

All that's left is the Select label, which reads:

```
Label (select)
Display (Off)
End of Line (Select)
Copy
Left ()
Display (On)
Return
```

which I designed to select and copy, quickly and unobtrusively, enough text so that I could find the position later on. I started with selecting five words to the right of the insertion point, but that was slow. Also, if in the process of counting those five words, the macro crossed a hard return, tab or other formatting code, that code was left out of the global (which contains only text). When I then went to find the text with the Go To macro, it didn't work since the Find command *does* recognize returns and tabs. Ouch.

But **End of Line (Select)** turned out to be fine. Fast, and is less likely to bump into a problematic code. If the insertion point is one word away from the right end of the line, the global only gets that one word but, including the document name and page number, there's still something to work with. The documentation recommends that the user have the insertion point further to the left in the line when calling the Assign macro.

(A more recent version of these macros, what's now available for download, uses a more advanced means to select text while handling codes. We'll learn this in chapter 21.)

The **Return** command, as you remember, takes execution back to the line following the Call command. The Call/Return structure is worth its weight in gold, in terms of the amount of code you'd need otherwise.

The Go To macro

Now that we have ten global variables full of data, let's look at the other half of the engine: the macro that finds positions based on what's in the globals. This part is a little more complicated. Starting out is easy, though. What else – build a menu!

```
Menu (Var00;"Go to Previous Position";{GlobalVar13;GlobalVar14;GlobalVar15;
GlobalVar16;GlobalVar17;GlobalVar18;GlobalVar19;GlobalVar20;GlobalVar21;
GlobalVar22})
Case (Var00;{1;one;2;two;3;three;4;four;5;five;6;six;7;seven;8;eight;9;nine;10;ten};can-
cel)
Label (cancel)
Go (end)
```

giving us labels according to the user's choice. Let's go to label one:

```
Label (one)
SubString Position (Var01;"  (";GlobalVar13)
SubString Position (Var02;")  ";GlobalVar13)
```

SubString (Var03;1;**Var01**-1;**GlobalVar13)**
SubString (Var04;**Var01**+3;**Var02**-(**Var01**+3);**GlobalVar13)**
String Length (Var05;**GlobalVar13)**
SubString (Var06;**Var02**+3;**Var05**-(**Var02**+2);**GlobalVar13)**
Go (go to**)**

which is, indeed, more complicated. We can see, though, that it's just a mess of Substring, String Length and Substring Position commands to pick apart a global variable – GlobalVar13 through 22, as appropriate. What do we pick out?

Var01: a number indicating the position of the first of the two spaces before a left parenthesis (note figure 39 for a chart. For clarity, the spaces are replaced by boxes).

Var02: a number indicating the position of a right parenthesis followed by two spaces.

Var03: a string representing the part of the global starting at position 1 (the first character), and going the length of Var01 minus 1 – to the last character of the document name. The document name is thus going into Var03.

Let's make sure we understand this. Figure 39 shows a chart of the surgery we're going to do on GlobalVar13:

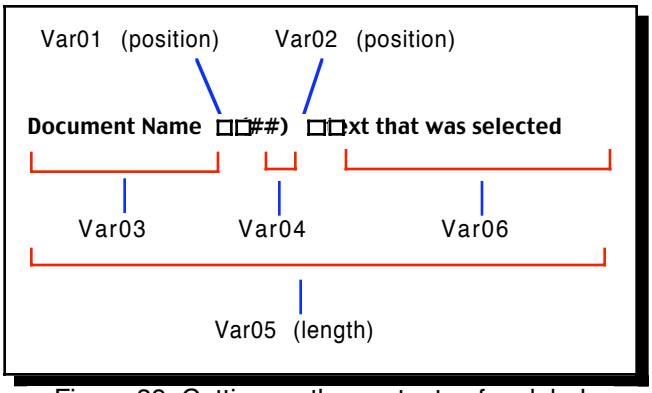


Figure 39: Cutting up the contents of a global variable

In the example above, with a file called "Document Name," the first special space is in position 14. The name of the document we're going to look for is thus contained in the substring going from position 1 for a length of 13, or Var01-1, in GlobalVar13.

Var04: starting with the position of the first special space plus 3 and going from there a length of Var02 minus the quantity Var01 plus 3, to give the page number. This allows for a varying number of digits in the page number. Note that Var04 is a string at this point, since it has been produced by the Substring command. We'll coerce it to a number when necessary.

Var05: the length of the global.

Var06: starts with Var02 plus 3, to give the first character of the text we'll search for, and going the length of the whole global (Var05) minus the position of the right parenthesis plus 2, to subtract the document name and page number.

Well, I told you this would be complicated. *Count this stuff out*, since you're going to miss the largest part of the value of this chapter if you don't see how the global variable is cut into local variables.

Why do any of this? Why not just use more globals? Nice idea, but there are only 50 of them, and they last for the whole current session of WP. And other macro mavens don't want you tromping all over their globals.

The information highway

OK, we now have all this information: Var03, Var04 and Var06, and can simply look for it among open documents. This is just cleanup. As you saw above, each label cuts up the global and then goes to a label called **Go To.** This consists of:

```
Label (go to)
On Error (error)
Select Window (Var03)
String To Number (Var04;Var04)
Go To (Var04;Current;No Change)
```

to select the document name contained in Var03, and go to the page number contained in Var04 – once we've coerced that string to a number, since WP can't count to the number of a page with a string.

We then want to search for the text in Var06, so we do this:

```
Select Page
Find/Change Direction (Within Selection;No Wrap)
Find/Change Where ({All})
Find/Change Match (Partial Word;Case Insensitive;Alphabet Insensitive;CharRep
Insensitive;{Text Only})
Find/Change Action (Select Match)
Find String (Var06)
Find
```

selecting the page first, so as to limit the range of our search.

What if?

If that search doesn't find anything, there could be two reasons: the text has moved to another page, or has been deleted. Another error will occur if the document can't be found among those open. This is a good place for you to download the finished macro from either address given in Appendix D, and look through the code. As you become more advanced at programming, you'll find yourself more interested in working samples than in explanation.

Task keys

I originally envisaged the Assign macro as being called by OnStartUp, running automatically every time you clicked the mouse within the document window. That works fine except that the macro is paused whenever you're not clicking the mouse and, with the present program version, screen redraw is not as good when a macro is paused. Also, running another macro containing an End Macro command will terminate that macro *and* any other that's paused.

You can try it like this, though, with a couple of quick additions.

First, add these lines to the top of the Assign script:

```
Label (top)
Pause Until (#Click#)
```

and add this line to the end of the script:

```
Go (top)
```

The **Pause Until** command is new to us. It pauses macro execution until it sees a character or **task key** typed. A task key can be anything listed in Preferences/Keyboard/Commands. Enclose the command in pound signs – WP's task key delimiter – in the script to designate it as a task key.

Another example of the usefulness of task keys is a macro I wrote to change the font to Symbol for a few characters:

```
Assign (Var00;FontName)
Font Name ("Symbol")
Pause Until (#Enter#)
Font Name (Var00)
```

so that the macro changes the font to Symbol and pauses until the user presses Enter, at which point the font is set back to whatever it was originally.

20: Odds ’n Ends

As you probably know, you can dig around in a document pretty effectively by opening the codes window. As a user, you can also leave that window alone. As a programmer, though, you can do amazing things with the codes.

Knowing the code

Say you have a document with three different fonts. You want to change one of the fonts from Palatino to Times, and leave everything else alone. The font definition unfortunately isn’t in a style sheet, so that won’t help. This macro will:

```
Codes (Show)
Find Next Code (Forward;Font Change) ; the code name is spelled exactly as you see it
    in the Find Code dialog
If (FontName="Palatino")
    Font Name ("Times")
    Delete Right ; this deletes the Palatino font code as long as the codes window is
    open
End If
Codes (Hide)
```

Note that the two-word “Font Name” is a command, while “FontName” all together is a read-only variable. All WP read-only (r/o) variables are named with one word.

You can put this script in a Repeat/Until (!FindStatusFlag) loop to fix an entire file, since FindStatusFlag reflects a code search as well as a text search. Turn display off for a big speed enhancement, since the codes window needs to be open for the Delete Right command to affect a code rather than a character.

Run the macro at the top of your file, as finding codes doesn’t have a wrap-around option. You can send execution to the top of your file with **Home ()** or **Goto Top of Document**. The latter puts the insertion point before any codes.

If you want to get elegant and test first whether the codes window is open, and restore that state when the macro ends, you could start the script like this:

```
If (ShowCodesFlag) ; if the codes window is open
    Assign (Var01;1)
End If
    Codes (Show) ; does nothing if codes window is already showing
```

and end the macro with:

```
If (!Var01)
Codes (Hide)
End If
```

Important design concept here: the macro looks for a generic font change code, and then tests for the specific font name. In this way, you can search for all sorts of stuff in a file, by first looking for the *type* of thing, and then testing for a more exact match.

Fonts should of course be put in style sheets in the first place, but what if you wanted to replace one style itself with another? Style sheets won't help there, but my friend and macro genius Dave Moulton did just this with a script written to find the code **Style On**, followed by a test with the variable **CurrentStyle** and then the command **Apply Style**, in his macro "Manipulate Styles," found on Corel's ftp site.

But I won't analyze the whole thing here, because a habit you'll want to form to become truly adept at macros is to look at the script of any macro you find whose design or operation interests you. Some scripts seem daunting at first, but you'll see that they're made up of parts – labels, loops and such – and the operation of each part isn't hard to understand. Flowchart a script as you read it, and the overall structure will become easy to see.

HTML codes

WP 3.5 added HTML editing. HTML styles are separate from WP's regular style sheets, a fine idea except when you want to take someone's file full of regular styles and translate them into HTML. This sample code assumes that your source file has styles "Heading1," "Heading2," and "Heading3," which you want to set to HTML headings 1, 2 and 3:

```
Goto Top of Document ; before any codes
Repeat
    Find Next Code (Forward;Style On)
    If (CurrentStyle="Heading1")
        HTML Heading (1)
    End If
    If (CurrentStyle="Heading2")
        HTML Heading (2)
    End If
    If (CurrentStyle="Heading3")
        HTML Heading (3)
    End If
Until (!FindStatusFlag)
```

So the paradigm with codes is, again, find the generic code and then test for the specific style, font or whatever.

Catching errors

Given how easy it is to include quote marks where you shouldn't, etc., it's nice that WP has error-checking for macro scripts. You've probably already seen the first level of error correction: when you save a script, WP **parses** it, looking for bad syntax, misspelled commands, and the like. It underlines what errors it finds, and requires you to fix them before it will save the script. You can save anything as text, of course. Or, if you're finishing up for the day and can't figure out what's wrong with one command, you can "comment it out" – put a semicolon in front of it, to turn the command into a comment line, save the script, and leave it. It'll be there for you tomorrow.

If your script parses OK but still doesn't run right, WP has three **debugging** commands to help you sort it out:

Step puts up a window showing the macro command that will run next, after you press any key. You can thus step through your script one line at a time, until you find the one that's going kablooey. The syntax is:

Step (On)

and you can change the **On** to **Off** and leave the command in the script, if you're going back and forth as you debug. Or turn step on only for the troublesome part of your code.

Speed slows down macro execution, by adding a delay between the execution of each line of code. The delay is expressed in 60ths of a second, so:

Speed (60)

will put a one-second delay between each command.

Wait does the same thing but only at the point where it occurs in the script. So:

Wait (180)

puts a single three-second delay into execution, while Step and Speed set the pace of things for the rest of the macro, or until another Step or Speed command is encountered. You could thus step through one troublesome part of your script. Wait is also useful when you put up a prompt, one of the ways of talking with your user.

Take a look in the online macro help, and you'll see that parameters for Prompt include the vertical and horizontal coordinates in points/pixels for the message, then the title, then the message itself. So an example might be:

```
Prompt (75;125;"Sample Prompt";"Are you awake right now? How do you know?")
Wait (180)
End Prompt
```

If a Prompt command isn't followed at some point by an End Prompt command, the prompt stays on screen until the user clicks in the close box (or a macro posts a different prompt). Compare that to an Alert, where, a modal dialog appears at the center of the main monitor and has an OK button. Menus give the user any number of choices, and Confirm posts multiple buttons on the order of OK/Cancel or Yes/No/Cancel. Only a prompt lets you specify its location, and only a prompt can display while the macro (or you) is doing other things.

My favorite place for a prompt is at the horizontal center of the screen, and about a third of the way down from the top. This is the Macintosh user interface's dialog position, more or less:

Prompt (ScreenSizeH/2-240;**ScreenSizeV**/3;**Var01**;**Var02)**

Pause can be used for debugging, or for the user's convenience. It's not the same as Pause Until. That command pauses the macro until a specific keystroke occurs, and then resumes execution. Pause simply stops the macro until the user chooses the **Continue** command from the macro menu. This can be useful in a number of situations, training for one. Imagine that your macro shows some steps in formatting, and then types some plain text for the user to practice on. The macro then pauses for the user to repeat those steps, after which he or she chooses Continue.

When we want an error

On occasion we plan for an error to occur at some point. I call the following script from my OnOpenDocument macro. If I've launched WP and have an empty Untitled document on-screen, and I open an existing document, this macro closes the Untitled doc automatically. It makes an exception and does nothing if the front window is either of the two that my OnStartUp macro calls. Glossary File assigns glossary entries as we learned in chapter 10; Data sets Citation format, as we saw in chapter 16. You type the **or** operator with shift-backslash.

```
On Error (cascade)
If (FrontWindow="Glossary File"|FrontWindow="Data")
    Go (end)
End If
While (Var01=Var02) ; or even While (1) - see below
    Select Window ("untitled*")
    If (!DocumentModifyFlag)
        Close
    End If
End While
Label (cascade)
Cascade Windows
Label (end)
End Macro
```

I wanted this to work for multiple untitled and unmodified windows, so I put in a **While** loop, specifying while Var01=Var02 (see below for more discussion of the While command). Var01 is always going to equal Var02 here, since neither is ever assigned a value. So the while loop will repeat forever *or* until the macro doesn't find any more windows with Untitled as the first part of their name. Then it returns an error. That's why the first line in the script is an error handler, sending execution to the Cascade label. This gets execution out of the while loop, with a guarantee that there are no more new and untitled windows.

On other occasions, you might want an ostensible error to occur at a specific point. The command **Return Error** does just that. **Return Cancel** is the equivalent of the user's pressing Command-Period at that point in script execution: if there's an **On Cancel handler** before that point, execution branches to the label it specifies. Otherwise, the macro quits.

Other error handling

Often enough, we're not looking for an error, but want to send execution in the right direction if one occurs. That direction can change depending on where we are in the script. The sample below is the "Save plus Editors" component of my Note Editor macros, version 3, an example of letting the macro do the thinking. What it's designed for is saving a footnote editor or an endnote editor (or both, if both are open) and saving the editor's paper – but not other open documents.

When the macro starts, it doesn't know if there's an endnote editor or a footnote editor open. So it tests for one by trying to select a window with, for example, "footnotes" as part of its name. If there is no such window open, WP returns an error. If there were no On Error handler earlier in the script, you'd just get an alert informing you of the error, and the macro would quit. But with an error handler, execution will go to a specified label if an error occurs. Thus, in quasi-macro terms:

```
On Error (Buy Plane Ticket)
Ask boss to double your salary
; more code
End Macro
Label (Buy Plane Ticket)
; code here will execute if boss threw you out the window
```

and here's some actual code:

```
If (NewDocumentFlag) ; untitled doc has never been saved
    If (!DocumentModifyFlag) ; because there's nothing in it
        End Macro ; nothing to do
    Else ; if anything is in the doc
        Save ; posts the dialog
        End Macro ; that's it for us
    End If
End If
Assign (Var01;FrontWindow) ; we need to remember which window is in front when the
macro starts
SubString Position (Var02;">~Footnotes";FrontWindow) ; if we get anything here, the
front window is a footnote editor
SubString Position (Var03;">~Endnotes";FrontWindow) ; or here, an endnote editor
Call (prompt) ; last subroutine in the script checks for unsaved changes in front window,
posts prompt, saves, ends prompt
;
If (Var02|Var03) ; if we have any kind of editor in front
    Assign (Var04;DocVar7) ; get name of paper from a doc variable in the editor
Else
    Assign (Var04;Var01) ; if an editor isn't the front window, then the paper is. So this
    if/else statement puts the paper name in Var04 in any case
End If
String Length (Var05;Var04) ; how long is paper name?
If (Var05>18) ; if longer than 18 characters
    SubString (Var06;1;18;Var04) ; get first 18
    Assign (Var07;"<"$Var06$"…>~") ; add carets, ellipsis and tilde per our naming
    convention
Else
    Assign (Var07;"<"$Var04$">~") ; just add carets and tilde if paper name is short
    enough
End If
;
If (Var02) ; if footnote editor is in front
    On Error (paper) ; go deal with the paper if you can't find an endnote editor
Else ; if anything but a footnote editor is in front
    On Error (footnotes) ; go looking for that footnote editor when you're finished here
End If
Select Window (Var07$"Endnotes") ; if there's an endnote editor, get it. If not - returns
an error - go to 'paper' or 'footnote' labels
Call (prompt) ; save the window
If (Var02) ; if a footnote editor was in front when the macro started (it won't be now if we
found an endnote editor to save)
    Go (paper) ; that footnote editor was then saved as the first step - saving the active
    window, so we don't need to select it and save it again, as the next subroutine would
    do
```

```
End If
;
Label (footnotes) ; from the error handlers for Var02, if the footnote editor wasn't in front
On Error (paper) ; we need another error handler in case the next command, to select a
window, returns an error - because there wasn't a footnote editor open
Select Window (Var07$"Footnotes") ; get footnote window
Call (prompt) ; save it
;
Label (paper) ; error handlers get us here if there's no footnote editor open when we look
for 1) an endnote editor, or 2) a footnote editor
Select Window (Var04) ; get the paper - or the original front window
If (Var02|Var03) ; if the original front window was an editor, the paper hasn't been saved
    Call (prompt) ; save it
End If
;
Label (prompt)
If (DocumentModifyFlag)
    Prompt (ScreenSizeH/2-240;ScreenSizeV/4;"Saving";"Saving changes to “"
    $FrontWindow$"”") ; I like this. Both Lotus 123 and More have a saving prompt
    Save
    End Prompt
End If
Return ; we got to this subroutine with the Call command
```

Doing our own error checking

Sometimes we wish a command would return an error when it doesn't do what we want, but the command is instead just ignored. Set Directory is one of those. I have an automated backup system on my Mac, using a Zip drive named "Backtrack" and the MacTools backup program. I have WordPerfect change the folder names on the Zip drive at every startup, appending the date (in "6/1/97" format) to any folder it finds named "Backup." So the Zip disk's window looks like figure 40:

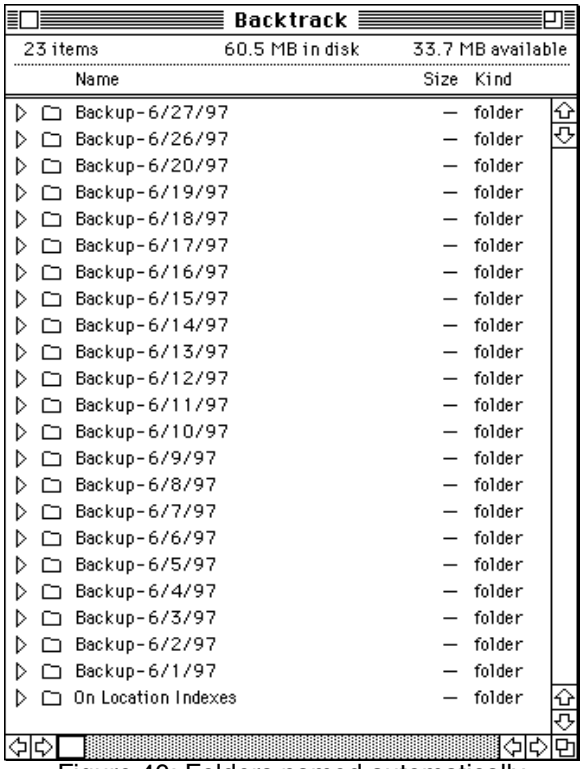


Figure 40: Folders named automatically

so that no backups will be overwritten, and a particular iteration of any work in progress will be a little easier to find.

The basics are easy enough. I open the trusty Data file in the System:Preferences:WordPerfect folder and get the text date (which I'll update before closing the file). That code looks like:

```
Prompt (ScreenSizeH/2-240;ScreenSizeV/4;"Backup Folder Date";"Checking backup folder date . . .")
Display (Off)
On Error (end)
Open Document (BootDir$"Preferences:WordPerfect:Data")
Position To Cell (1;3;1)
Select TableCell
Copy
Date Format ("[Month#][Date Sep][Day#][Date Sep][Leading Zero][Yr]")
Date Text
Save
Date Format ("[Month(Abbr)] [Day#], [Year]")
Close
```

That leaves a table cell containing the date on the clipboard. Depending on where we use the clipboard, the code for the table cell may simply disappear, or may cause an unexpected result, such as becoming a hard return in some pastes. Look at the clipboard to see what I mean. Best to get rid of the final character – the cell code – with:

```
String Length (Var01;Clipboard)
```

```
SubString (Var02;1;Var01-1;Clipboard)
```

giving us just the date in Var02.

Now comes the tricky part. We can assume that the Zip disk has a folder entitled just "Backup" and do this:

```
Rename Folder ("Backtrack:Backup";"Backtrack:Backup-"$Var02)
```

but that will return an error if this macro is run since the last time the backup program has run, so there's no folder without a date. Not a large problem, but the error is presented as an alert, which the user has to OK. How can we check whether this dated folder exists?

I used the **Set Directory** command to go to that folder, if there is one, with:

```
Set Directory ("Backtrack:Backup-"$Var02)
```

except that this command doesn't tell us anything if there is no such folder. So let's check whether that command did what we wanted, with:

```
If (CurrentDir="Backtrack:Backup-"$Var02$":")
    Go (end)
```

and here I had to work around a quirk with command syntax – a problem only because I wasn't expecting it. As you know, path descriptions on the Macintosh are composed of disk, folder and file names separated by colons. For the Set Directory command, the final colon in the path is assumed, but for the CurrentDir variable it's not! I had no idea what I was doing wrong until, error checking, I inserted a line putting CurrentDir in an alert, noted the final colon, and revised the line above to have that final colon.

At that point I could do things if the dated folder wasn't found, like see if there's a non-dated folder extant. I found that adding the final colon to the Set Directory parameter worked, so I decided to standardize:

```
Else
    Set Directory ("Backtrack:Backup:")
    If (CurrentDir="Backtrack:Backup:")
```

If there's such a non-dated folder, add the date:

```
Rename Folder ("Backtrack:CP Backup Files";"Backtrack:CP Backup Files-"$Var02)
```

Otherwise, are we not finding a folder because the disk isn't in the drive? Let's set the directory and then check it, for just the volume name:

```
Set Directory ("Backtrack:")
If (CurrentDir!="Backtrack:")
    ; tell user to put the disk in the drive
End If
```

and that takes care of the options. Here's the whole script:

```
Label (top)
```

```
Set Directory ("Backtrack:")
If (CurrentDir!='Backtrack:')
    Prompt (ScreenSizeH/2-240;ScreenSizeV/4;"Backup Folder Date";"Missing backup
    volume! Check Zip drive, insert disk "Backtrack," and press Enter. Or, press
    Command-Period to end Backup Folder Date macro.") ; since prompts show in the
    Chicago or Charcoal fonts, you can use "⌘" instead of "Command."
    Pause Until (#Enter#)
    Go (top)
End If
Display (Off)
Prompt (ScreenSizeH/2-240;ScreenSizeV/4;"Backup Folder Date";"Checking backup
folder date . . .")
;
On Error (error)
Select Window ("Data")
Go (start)
Label (error)
Open Document (BootDir$"Preferences:WordPerfect:Data")
Assign (Var49;1)
Label (start)
On Error (end)
Position To Cell (1;3;1)
Select TableCell
Copy
String Length (Var01;Clipboard)
SubString (Var02;1;Var01-1;Clipboard)
Date Format ("[Month#][Date Sep][Day#][Date Sep][Leading Zero][Yr]")
Date Text
Save
Select TableCell
Copy
Date Format ("[Month] [Day#], [Year]")
If (Var49)
    Close
End If
String Length (Var03;Clipboard)
SubString (Var04;1;Var03-1;Clipboard)
If (Var02=Var04)
    Go (end)
End If
;
Set Directory ("Backtrack:Backup-"$Var02)
If (CurrentDir="Backtrack:Backup-"$Var02$":")
    Go (end)
Else
    Set Directory ("Backtrack:Backup:")
    If (CurrentDir="Backtrack:Backup:")
        Rename Folder ("Backtrack:Backup";"Backtrack:Backup-"$Var02)
    End If
End If
Label (end)
End Prompt
Set Directory ("Bebop:Education:") ; ready to go to work
Display (On)
```

and BTW, that Zip disk goes off-site at the end of the month, to be replaced by another, rotating, "Backtrack." A good way to avoid losing all those neat macros you're starting to write.

Accessing variables

Often enough, while coding a macro we need to see what's in a given variable – either a read-only or read-write. So WP lets you assign keystrokes to variables, just like assigning keyboard equivalents to commands and menus. Go to Keyboard in Preferences and, from the pop-up menu at the top left, choose Variables. You get a list of every variable in the program. The keystroke will type the value of the variable at the insertion point, and you can check its value at your convenience. Among the read-write variables, this works best for global, document and script variables, since local variables lose their value when the macro containing them ends. I usually put local variables in an alert to check their value. Note that alerts don't display numbers, so if Var01 has a number in it, an easy way to coerce it to a string is by:

```
Alert (' '$Var01)
```

Let's get loopy

What's this with the **While** loop on page 118? It's much like **Repeat/Until**, but can be more efficient in some situations. In the macro above, I could have had line six as "Repeat" and line 13 as "Until (!Var01=Var02)" – same difference. Since neither Var01 nor Var02 have been assigned anything, the While command will always be true, and the Until command would never become true – so the macro keeps looping until there's an error. Since Repeat/Until and While differ in the nature of the test, one or the other may be better suited to a particular purpose.

A third kind of loop WP offers is **For.** This is a little more sophisticated, with the increment that's going to affect the test included in the command line. The four parameters are: variable, initial value, test, and increment. So:

```
For (Var01;0;Var01<Var49;Var01+1)
     ; some code here
End For
```

will assign Var01 a value of 0, and test whether Var01 is less than Var49. If it is (the result is true), Var01 is then assigned the fourth parameter (in this case, Var01 is incremented by 1), the code between the For and End For commands is run, and execution returns to the For line. Here's an example of the three types:

```
Assign (Var49;25)
;
Menu (Var00;"For/Repeat/While";{"For";"Repeat";"While"})
Case (Var00;{1;for;2;repeat;3;while};cancel)
;
Label (for)
For (Var01;0;Var01<Var49;Var01+1)
     Type (Biff )
End For
End Macro
;
```

```
Label (while)
While (Var01<Var49)
    Type (Biff )
    Assign (Var01;Var01+1)
End While
;
Label (repeat)
Repeat
    Type (Biff )
    Assign (Var01;Var01+1)
Until (Var01=Var49)
End Macro
```

The choice can result in a line or so less code, here and there. More importantly, one structure may, in a given situation, allow you to build a script that is clearer conceptually to you. That's what counts.

Another type of loop is offered by **For Each**. Here, instead of incrementing a variable, the several values of the variable are included in the command line, and the sequence loops for each value. Thus:

```
For Each (Var01;{'Alpha';'Beta';'Charlie'})
    Type Var (Var01)
    Hard Return
End For
```

which can often be a space-saving convenience.

Loops at the 'macro' level

We can take this loop/test idea up a level, and have one macro execute another if a certain test is met, using either the Run command, to execute that other macro right away, or **Chain,** to run the other macro when the first macro is finished. If you use the Run command, execution returns to the first macro when the second is done – unless execution encounters an **End Macro** command. This is a primary use of this command, since macros end anyway when they run out of code.

Why not put all your code in one macro? There's more than one good reason! Consider: *size*. My Character Styles is a set of 28 macros, totalling over 3000 lines of code. Even if it added up to less than 32K, the limit for a macro, it would be crazy to try and edit all that code in one script. This is partly since WP's macro editor doesn't have Find/Change or other features that overall use wouldn't justify, but mainly because a manageable size is vital to *good organization*. Within a single macro, there's lots of places to put this or that little tidbit of code: initialize a variable, set a font, whatever. All those little hiding places make them harder to find when you decide to change them. With a set of separate, modular and interacting macros each with a more specific purpose, editing is easier. You can also save generalized versions of these single-purpose macros in *libraries*, which make easy building blocks for future projects.

Starting to put things together

At this point we've covered a great deal, but macros may seem more mystifying than ever. Even if the examples made sense right away, you still might be wondering, "How will I ever learn what all those variables are?" and "When will I start to grasp the big picture?"

When you think of it, questions like these are what everyone has when learning a language: Swedish or Thai as well as macros. There just seems to be too much to digest. But don't worry. Learning is a funny endeavor: it rarely takes a linear pace. Everything seems overwhelming for a while, and then – presto – parts start to fall into place.

Let's help this along with some learning strategy. Like any language, the WP macro language has two basic parts: grammar and vocabulary. Grammar, or the If/Else, Repeat/Until kinds of structure we've learned, is best approached by example and practice. Vocabulary, though, can be learned either this way or by browsing WP's online macro help (not the standard help file, but "WP Macro Help," the fourth choice on the balloon help menu). Otherwise, a printable list, separated into commands and variables, is included in the program's ReadMe files.

Overall, the most effective way to learn code is by studying examples. Now that you know how to troubleshoot and write tight loops, we'll spend the next few chapters dissecting some sophisticated and elegant code. Those adjectives will, in short order now, describe the macros *you* write.

21: Outlining

A structured outline was a great way to write before the advent of the personal computer, and is orders of magnitude better with a computer. You can now create an outline and then move parts of it around for more effective organization, hide subtopics for an overview, add speaker's notes, and do about twenty other things. Then, convert the outline to text and you're done, or modify the format for presentation.

WordPerfect's outlining feature will let you do most of this, but I saw a few things I wanted to add. Indentation, for example, is an aid to easy comprehension of an outline. As the program ships, outlining goes like this:

I.This is the first topic. There's no automatic space between the topic's label (the Roman numeral one in this case) and the beginning of the text.

A.This is the first subtopic. With Outlining mode on, I pressed a Return and then Tab, and the Roman numeral II changed to a capital letter A.

1.This topic at the third level was generated by pressing Tab twice. Note that at any level, whole topics are not indented – the second and succeeding lines return to the left margin.

B.This topic should be at the second level, so I only pressed Tab once. The writer has to keep track of which level he or she is on.

While outlining formats often have indented topics, like:

I. By contrast, this topic's label is separated from its text by one tab stop. The entire topic is indented. Considering each topic as an idea, this formatting makes the writer's thinking easier to follow.

A. This first-level subtopic is likewise fully indented. It's true that there is a trade-off between visual clarity and the number of words that will fit on a page.

1. And so on for each level in the outline.
2. When I pressed Return at the end of the previous topic, the outline generated a new topic at the same level. The user doesn't have to count Tabs.

So I wrote a set of macros to produce the second kind of outlining in WordPerfect. The macros also let you move whole topics around with a keystroke and click, cross-reference topics, clone and gather topics, generate a table of contents, collapse and fold topics, and several other things – in fact, as many as the best dedicated outlining programs.

Let's look at this set and see, for starters, how the macros produce the basic difference in formatting that you see above.

Outline Return

Part of the design strategy would be to turn the program's Outline Mode on when creating topic labels, but leave it off otherwise since WP's QuickCorrect doesn't work with Outlining Mode on (since outlining label punctuation would cause untoward effects). QuickCorrect is a lifesaver for me and I wanted to keep it (and in fact added to it in outlining, to capitalize the first word of each topic).

So the Outline Hard Return command involves typing one keystroke that turned outlining on, added a carriage return (thereby adding a topic label), indented, and turned outlining off. Straightforward stuff, and the code then starts out as:

```
Outline Mode (On)
Hard Return
Indent
Outline Mode (Off)
```

which does basically what we want. This code puts every topic at the left margin, though, so that every label starts out as (in default formatting) a Roman numeral. Dedicated outlining programs do it differently: they maintain the level the user is working at. That will be harder to do but, hey, we're pros at this.

We'll use the LineCharacterCount variable, which tells us how many characters are on the current line to the left of the insertion point. We'll search backwards for the preceding outline label code, called **Paragraph Number**, and see how many characters it is from the left margin. When we search for and find a code, the insertion point lands just to the right of the code. So one character in LineCharacterCount will be the Paragraph Number code. Any other characters would be Tabs, so we'll just insert Tabs to match LineCharacterCount minus one, and we get:

```
Outline Mode (On)
Hard Return
Left () ; to go to left of the label we just created
Find Next Code (Backward;Paragraph Number)
Assign (Var01;LineCharacterCount-1)
Find Next Code (Forward;Paragraph Number) ; to get back to the label we just created
For (Var02;0;Var02<Var01;Var02+1)
    Tab
End For
End If
Indent
Outline Mode (Off)
```

To test this script, run it and type a few words for the first topic; then run it again. So far so good, but we're operating at the left margin. So run this test:

1. create a new topic

2. type a few words

3. put your insertion point to the left of the label

4. press Tab, to move the label to the right and change it to a capital letter

5. put your insertion point to the right of the text you typed

6. run the macro again

You'll get a new topic, indented one level like the topic above it. Now all we need is an Outline Tab command (and a similar Back Tab command).

Outline Tab

As you can see by working with the stock outlining protocol, if Outline Mode is on, pressing Tab when just to the right of a label tabs the label. This works fine with stock outlining, since a Return leaves the insertion point right next to the label. We've added an Indent to our Outline Return, though, so the standard setup won't work for us.

What we can do is find the preceding Paragraph Number code. That find puts the insertion point to the immediate right of the code, at which point a Tab will do what we want it to, as long as Outline Mode is on. I've added an End of Line () command at the start of this segment, just so that Outline Tab will do what it should no matter where in the line the insertion point may happen to be. For example, the user might have typed several lines past this one, and then decided that this line should be indented one level further. He or she clicks to the *left* of the label, hits Outline Tab, and presto – the line *before* this one gets indented further. What else could happen, given that we're looking backwards for a Paragraph Number code? Sometimes, to write good code you have to stay one step ahead of the user.

After the Tab, I've added a Right () command, to move the insertion point from the Paragraph Number code past the Indent that our Outline Return command put there. If the user hasn't typed anything in that topic, the insertion point is just where it should be. So, for an Outline Tab command, we have:

```
End of Line ()
Find Next Code (Backward;Paragraph Number)
Outline Mode (On)
Tab
Right ()
Outline Mode (Off)
```

Now let's do a Back Tab command, and our basic outlining engine will be done! Back Tab is of course useful when the user decides he or she wants a particular label less indented – and this will be a common choice, given that our Return command maintains the level of indentation.

To get a clearer picture of what we need to do, let's see what an outline looks like in the Codes window, as in figure 41:

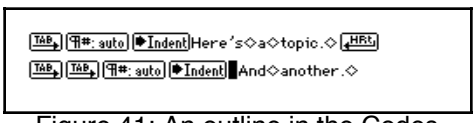


Figure 41: An outline in the Codes window

where we see the first topic tabbed one level (one Tab at the left of the line), and the second topic tabbed one level further. Our user wants to move the second topic left by one level. For scripting purposes, we don't know where the insertion point is, except that we can assume it's in the topic the user wants to move. So we need to start with an End of Line () so that this will work anywhere in the line, and then search backwards for the Paragraph Number code (shown as "¶ #:auto" in the Codes window).

Then go left a character, since the Find Code command puts the insertion point to the right of the code it finds. Then, delete a character to get rid of one Tab. Then, go right two characters, past the Paragraph Number and the Indent, to the beginning of the text entry area. And we have:

```
End of Line ()
Find Next Code (Backward;Paragraph Number)
Left ()
Delete
Right ()
Right ()
```

A more advanced operating mode

What we have so far is a basic outliner with three commands, which the user could assign specialized keystrokes to, and that would be that. What I wanted, though, was an outliner that would use the Return, Tab and Shift-Tab keystrokes for *both* outlining and regular text entry, making the user's life much easier.

To do this sort of thing, we set up a global variable as a flag, and tell our outline macros to look at it. If the variable contains a value, do the Outline Return or Outline Tab code; otherwise, do a regular Return or Tab. Let's use GlobalVar23, and add to the macros as follows.

```
If (GlobalVar23) ; if there's something in GlobalVar23
    Outline Mode (On) ; this is the program's outline mode
    Hard Return
    Left () ; to go to the left of the label we just created
    Find Next Code (Backward;Paragraph Number)
    Assign (Var01;LineCharacterCount-1)
    Find Next Code (Forward;Paragraph Number) ; to get back to the label we just
    created
    For (Var02;0;Var02<Var01;Var02+1)
        Tab
    End For
    Indent
    Outline Mode (Off)
Else ; if there's not anything in GlobalVar23
    Hard Return ; the standard keyboard command
End If
```

and assign this macro the Return key *itself*. We can do the same for Outline Tab and Outline Back Tab (i.e. Shift-Tab).

We now have an outline mode that needs to be turned on and off. This short macro will do that, by changing the value of GlobalVar23.

```
Menu (Var01;"Outline Mode";{"On";"Off"})
Case (Var01;{1;On;2;Off};Cancel)
Label (Cancel)
End Macro
Label (On)
Assign (GlobalVar23;1)
End Macro
Label (Off)
Assign (GlobalVar23;0)
```

For an added touch, you could add a third menu choice, to go to the Outlining Dialog. The command for that dialog is, simply, "Outlining Dialog." The commands for other dialogs are similarly straightforward. They're listed, of course, in the online macro help.

Otherwise, you could tighten up that code by changing it to:

```
Menu (Var01;"Outline Mode";{"On";"Off"})
Assign (GlobalVar23;2-Var01)
```

And there we are! The start of a complete outlining facility. There's one drawback to our design: if you use the Return and Tab keys to navigate dialog boxes (e.g. the Return key to click the heavily-bordered button), you'll want to use the Enter key instead. This is because all macros pause when dialogs are open, so Outline Return, although it delivers a regular hard return handily in a document window when Outline Mode is off, won't have any effect when a dialog is open. The Enter key is fine, or you can assign any other keystroke to the Hard Return command in Keyboard in Preferences. Same with Tab and Back Tab.

Dragging and dropping topics

Note that if you want to drag topics around to reorganize, turn on Drag and Drop in Environment Preferences, then triple-click to select an entire topic (paragraph), dragging further to select any subtopics. Then release the mouse, click again and drag the insertion point to the left margin of the topic you want your dragged topic to land above. Presto. But we'll have a fancier way of doing this, which will automatically move all subtopics as well.

22: Outlining part two

Last chapter we built a basic outliner. This chapter we'll make it fancy. In doing so, we'll learn how to script style sheets, how to maintain structure when tabbing a topic, and also how to use another kind of Find command – available only in macros – that spells power with a capital P. My kinda fun.

Doing it in style

Style sheets are a powerful feature in word processing, and controlling styles with macros reminds me of someone's ad about the power of two. What we'll look at here is assigning level-specific styles in outlines (so that level one looks different from level two, etc.) and also working with styles that effectively make text invisible, but you can use the concepts we explore to do any number of other things. You'll end up with what will seem like a home theater's remote control for your documents: press one button, and a bunch of things happen. Cool, and supportive of accuracy and speed.

I set up the outlining macro set with eight styles – one per level – called "zs1" through "zs8": the letter z just to put the styles out of the way at the bottom of the style menu (if the user wants to keep them on the menu in the first place), the letter s for style, and the number for the level. I also defined a style called "zc" – c for collapsed – that would be applied to text that the user wanted to hide, to present only the main topics in the outline. Macro commands will apply the zs styles or remove them (to get an outline in plain text) or apply the zc style to collapse whatever range of levels the user selects, so that only the main two or three levels are left visible – a good way to get the big picture.

The first thing we want to do when applying styles to paragraphs is make sure what amount of text we're treating. This is a setting of format orientation, discussed on page 32. The Character option proved difficult from the user-interface point of view, so it's been deleted from the Preferences menu. We'll find a use for it, though.

For most of style scripting, all we want to do is work with the whole paragraph containing the insertion point. This is the Single Paragraph orientation. We could just set it to that, but this would ignore, and possibly change, the user's setting, and that's bad programming. What we should do is record the user's setting, change it to what we need, do what we want, and then restore the user's preference.

The read-only variable is **FormatOrientation**, with values of 0 for Character, 1 for Paragraph and 2 for Single Paragraph. So our macros will start out with:

```
Assign (Var01;FormatOrientation)
Formatting (Single Paragraph)
```

and end with:

```
Case (Var01;{0;character;1;paragraph;2;single paragraph})
Label (character)
Formatting (Character)
Go (end)
Label (paragraph)
Formatting (Paragraph)
```

```
Go (end)
Label (single paragraph)
Formatting (Single Paragraph)
Label (end)
```

Note that the last code snippet includes a Case command which isn't preceded by a Menu command – something new to us. This is, however, fairly common use. A lot of what macros do is make choices depending on the case of things – whether the user has set that case with a menu, or otherwise. So these snippets put the current setting in Var01, and restore that setting at the end. In between, the Single Paragraph orientation lets us set styles for the current paragraph only, without selecting anything.

Now that we're able to apply styles to a single paragraph, how do we tell WP to assign e.g. style zs1 to level 1 and so on? Just two lines:

```
Find Next Code (Forward;Paragraph Number)
Apply Style ("zs"$LineCharacterCount)
```

so the macro searches for the next outline label, landing just to the right of it. The LineCharacterCount variable will then equal the level of that topic, and that topic receives the style it should. Go to top of document first and repeat until FindStatusFlag=0, and you've changed an outline from plain to styled text.

When styled text is in effect, adding a styled topic with Outline Return is even easier: the return in outline mode leaves the insertion point immediately to the right of the paragraph number, all ready for the Apply Style line of code.

Keeping tabs on things

An important feature of dedicated outliners is that they maintain structure of subtopics when you tab or back tab a topic. The macro can tell what counts as a subtopic with, again, this vital variable LineCharacterCount. We start by assigning it to a local variable and then testing LineCharacterCount at each subsequent paragraph number code. If the read-only variable is larger than the local, the macro inserts a tab before the paragraph number code. Otherwise, the macro realizes it's come to the end of the subtopics to be tabbed. And we have:

```
Assign (Var01;LineCharacterCount)
Repeat
    Find Next Code (Forward;Paragraph Number)
    If (FindStatusFlag)
        If (LineCharacterCount>Var01)
            Tab
            Assign (Var02;Var02+1)
        End If
        Right ()
    Else
        Assign (Var02;Var02-1)
    End If
    If (!FindStatusFlag)
        If (Var02>=1)
            Assign (Var02;Var02+1)
        End If
```

```
        End If
    Until (!FindStatusFlag)
```

and what's Var02 doing? It's keeping track of how many topics have been tabbed, so that the macro can count back that quantity of paragraph numbers as its last action – to return the insertion point to the topic it was in when the user pressed the Tab key.

The finished code contains more than this, largely lines to change the level-specific style of topics now at a new level, check whether a topic is being tabbed to the right of level 8, and such things.

The raw truth

Now, let's step into advanced programming. As you know, WP documents are composed of letters and of codes that do things with those letters. We can say that both letters and codes are objects or, since we're dealing with basic parts of a document, we can call them **raw objects**, and the **raw object type** of letters and codes differ. While it's often useful to do a find for a given character or string (using Find/Change) or for a given code (using Find Code), we sometimes want to read *whatever* is to the left or right of the insertion point and, based on what we find, make a choice.

What I wanted to do was give the user a way to bypass the tab command's automatic tabbing of subtopics. The user might have, for example:

> I. Some topic.
> A. Some subtopic.

With the insertion point at the end of the level 1 topic, the user then hits return, and gets another level 1 topic (II). What he or she wants, though, is another topic at level 2. If the user just presses tab, topic II becomes subtopic A – but everything subordinate to that subtopic gets indented a further level.

I could have added a separate command to tab only the current topic without affecting subtopics, but I wanted to simplify things, so that just pressing Tab will do what the user most likely wants it to. In this case, with no text in the topic yet, the user would want to position only the current topic.

So, let's set up a command sequence which does this: if there's text to the right of the insertion point, the Tab command tabs subtopics. If there isn't text to the right, it tabs only the current topic.

We start this with a **Raw Read** command to see what's to the right of the insertion point. Raw Read can read for anything, or for a character, or a function (code), for a WPChar (found in imports from DOS documents) or for a ScriptChar (a character in another script). Since what we want to determine is whether what's to the immediate right of the insertion point is a character or something else, we'll read for anything, going right, and our line of code is:

```
Raw Read (anything;Right)
```

which will find whatever's the next thing to the right of the insertion point, and put it in the **RawObject** variable. It will also put a code representing the type of object in the **RawObjectType** variable. Codes for possible types are:

Character = 1 (any character)
Function = 2 (most codes)
WPChar = 3 (codes which may import with WP DOS files)
ScriptChar = 4 (characters in a script other than the current one)

and, working with Mac documents in the Roman script, anything in a document is type 1 or 2. So,

```
If (RawObjectType=2)
```

there's a code, not text, to the immediate right, so we just tab the topic (and apply a style, if that option is in effect, and so on). This would be the case for a new topic in an existing outline, as in the example above. There's no text to the right of that topic, but there is a hard return to the right. If we had found text to the right, we'd tab all subtopics.

The user then has the further option of tabbing only the current topic, even if it contains text, by clicking to put the insertion point to the right of the text. If the user wants to tab subtopics, he or she can hit Outline Tab when the insertion point is anywhere else. Spiffy!

23: Outlining part three

Let's look at a few more in's and out's of the macros in my outlining set – less so that you can write your own outliner than to explore some fine points of getting WordPerfect to jump through the hoops *you* want it to.

A great advantage of a computer-based outline over hard copy is that you can move topics around to reflect reorganization and clarification of thought. WP has drag and drop, which you can use in an outlined document as well as anywhere else, but moving a topic (which the program sees as a paragraph) doesn't take its subtopics with it. In dedicated outlining programs, moving a topic (i.e. an idea) does take its subtopics (the details of the idea) with it. So let's get WP to do that too.

Moving topics

As you remember from last chapter, the LineCharacterCount variable is critical to much of outlining design. It tells us how far the insertion point is from the left margin, which in turn will tell us if the topic below the current one is a subtopic. The basic strategy to gather up all subtopics of the current one is: search forward for paragraph number codes, checking each one to see whether it's farther right (has a larger value in LineCharacterCount) than the number code contained in the topic we started with. If so, we increment a counter and continue. When we find a topic that is not farther right than where we started, we know that we've included all subtopics. We then count back the number of paragraph codes that we counted up, selecting as we go, and cut the selection to the clipboard.

Then, when the user clicks the mouse to tell us where he or she wants that topic to go to, we'll search for the beginning of the topic, and paste the clipboard just above it.

To elucidate the details, I'll comment the code, as follows:

```
On Error (end) ; any problems will take us to the end label which, since it's at the end of the script, will simply abort the macro
If (SelectionFlag) ; if the user has already selected topics to move, our task is much simpler
    Cut ; cut the selection and go to the paste part of the script
Else
    Find Next Code (Backward;Paragraph Number) ; are we in an outline topic? If not:
    If (FindStatusFlag=0)
        Alert ("Topic not found.")
        Go (end)
    End If
    Display (Off) ; just to speed things up. This can make quite a difference.
    Prompt (ScreenSizeH/2-240;ScreenSizeV-50;"Move Topic";"Preparing to move topic and its subtopics") ; let the user know what's going on, with a command that can take a moment to complete if there are several subtopics. The calculation of prompt location puts the prompt at the horizontal center of the screen, and near the bottom of the screen – most probably out of the user's way.
    Assign (Var01;LineCharacterCount) ; what level is the current topic?
    Repeat
        Find next Code (Forward;Paragraph Number) ; find the next topic
        If (FindStatusFlag=0) ; if the current topic is the last one in the outline
            Find Next Code (Forward;Return-Hard) ; find the end of the topic
```

```
            If (FindStatusFlag=0) ; if no hard return, we must be in the last paragraph of
            the document
                End () ; go to the end of the document
                Assign (Var03;Var03+1) ; we're counting paragraphs
                Hard Return ; add a hard return if there isn't one, to avoid problems
                when pasting.
                Go (Select) ; nothing more to count, so execution branches to the code
                that selects the number of paragraphs we've counted
            End If
            Assign (Var03;Var03+1) ; add this topic to the number we've counted
            Go (Select) ; we're done counting, so start selecting
        End If
        Assign (Var03;Var03+1) ; add this paragraph to the counter
        Assign (Var02;LineCharacterCount) ; get level of indent
    Until (Var02<=Var01) ; compare level of indent of the current topic with that of the
    topic we started with
```

and let's take a break from the code to make sure we're following all the nested If statements. Figure 42 shows a flowchart:

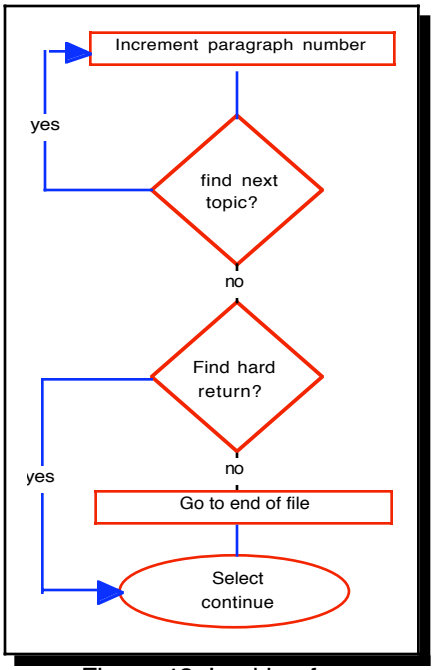


Figure 42: Looking for subtopics

showing how, just before the select label, we have 1) an insertion point that's as far down in the file as we want (includes all subtopics) and 2) in Var03, a count of how many subtopics that is.

We can now start counting back - decrementing Var03 as we go, until we're back at the topic we started in. I did this with the Find/Change command, looking for Hard Return codes. It's somewhat ungainly, but Find/Change lets you extend the selection as you go, unlike the otherwise

slicker "Find Next Code (Backward; Return-Hard)." (Raw Read would work, if you start a selection first – something like:

```
Left (Select)
Repeat
    Repeat
        Raw Read (function;Left)
    Until (RawObject=[Return-Hard Code])
    Assign (Var04;Var04+1) ; counting backwards
Until (Var04=Var03+1) ; until the one counter matches the other
```

which I didn't use because I only subsequently found out that Raw Read will extend an existing selection. I'm changing to this method in the next version). Anyway, what I used was:

```
Left ()
Beginning of Line ()
Label (Select)
Repeat
    Beginning of Line (Select)
    Find/Change Direction (Backward;No Wrap)
    Find/Change Where ({Current Doc})
    Find/Change Match (Partial Word;Case Insensitive;Alphabet Insensitive;CharRep
    Insensitive;{Text Only})
    Find/Change Action (Extend Selection)
    Find String ("[Hard Return]")
    Find
    Assign (Var04;Var04+1) ; counting backwards
Until (Var04=Var03+1) ; until the one counter matches the other
```

or until it almost matches the other. They should match but, when I wrote this, I wasn't getting the right result. I got that right result by adding 1 to Var03. This highly advanced programming concept, known as the "fudge factor," is extremely valuable when fixing code. It's also much easier than trying to figure out why the script wouldn't count right in the first place.

```
    Right (Select) ; delete the last hard return from the selection
    Cut
End If
Prompt (ScreenSizeH/2-240;ScreenSizeV-50;"Move Topic";"Click in topic below where
you want the moved topic(s) to appear.") ; no need to end the previous prompt first. This
one just replaces it.
Display (On)
Pause Until (#Click#) ; pauses macro execution until a mouse click. The pound signs
delineate a task key, which we learned in Chapter 18. This can be anything listed in the
Commands menu in the Preferences/Keyboards dialog. Click, or Hard Return or Enter
are common choices for this. Click is the necessary choice here, to locate where the
selected and cut topics should be pasted.
Find Next Code (Backward;Return-Hard) ; for maximum convenience, I let the user click
anywhere in the topic above which he or she wants to paste the moved material. So this
gets us from the click point to the end of the preceding paragraph/topic.
If (FindStatusFlag=0) ; if there is not a preceding paragraph, we're in the first paragraph
of the file
    Home () ; go to the top of the file
    Hard Return ; make some space to paste it
End If
Paste
```

```
End Prompt
Find/Change Reset ; restore the user's preferences
Label (end)
```

and there we go. Not too hard in basic terms, but a good example of how you have to make allowances for where the user might be in the document. Code that works in the middle of the document is quirky in the first or last paragraph.

A note about using the RawObject variable. You can test for any code listed in the Find Code dialog, spelled exactly as in that dialog, followed by the word "Code" and enclosed in brackets. But, if you did something like:

```
Raw Read (function;Left)
Assign (Var01;RawObject)
```

you'd get a number representing that code, in Var01. You could test for that number, but it's easier to test by name:

```
If (Var01=[Style On Code])
    Alert ('Biff')
End If
```

Appendix C has a list of the raw object code numbers. You could write the above snippet as:

```
If (Var01=-9471)
    Alert ('Biff')
End If
```

which, in some cases, works better than the code name. In some instances, a code value represents either of a pair of codes: Italics On and Italics Off, for example. Your script will then read until it finds the first of those.

Marking and referencing topics

Cross-referencing is a wonderful thing, but you have to take multiple steps to do it. Not unreasonable – the program doesn't know whether you're referencing a graphic, paragraph, page, text box, footnote or a few other things, so you provide that information in the List dialog.

When we know ahead of time what we want to mark or reference, though, we can write a macro to do most of the work. This illustrates a nice use of the macro feature: to automate a command you use often and which otherwise involves wading through a large dialog and making the same choices each time.

So, when in an outline, we can nearly automate the process of referencing another topic in the outline, and it's well worth it. "See topic X.A.2" is a helpful addition to things, especially when the references update on command. Let's see how to make cross-referencing within an outline as easy as possible. With a little editing, this could be useful for other kinds of cross-referencing you do. My Equation Macro in "John's WP Tips and Macros" shares this design to reference equations.

There are two commands: one to mark and one to reference. Marking is a simple task. Here's the code:

```
End of Line () ; to make sure the user doesn't mark the topic preceding the current one,
which will happen if the insertion point is to the left of the label.
Find Next Code (Backward;Paragraph Number) ; are we in a topic?
If (FindStatusFlag)
    Get Text (Var01;"Mark Outline Topic";"Enter an ID to mark this topic for
    cross-reference elsewhere:")
    If (Var01!="") ; has the user put anything in the Get Text dialog? Clicking OK or
    pressing Return with nothing in the text entry field will leave Var01 empty. This lets
    the user just press Return to cancel the dialog, if he or she has decided not to mark
    the topic after all. Users appreciate this kind of thing.
        Mark Target (Var01) ; the program command
    End If
Else
    Alert ("I can't find an outline label to mark.")
End If
```

which lets the user put a mark ID at the topic he or she wants to reference. The reference macro, in turn, is:

```
Get Text (Var01;"Reference Outline Topic";"Enter the ID used to mark the topic for
cross-reference:")
If (Var01!="") ; if we get anything in Var01
    Create Reference (Var01;Paragraph)
End If
```

which is not much more than the program command, with parameters for the ID and the type of reference. Of course, this command produces what looks like a question mark in your text, until you generate in the List dialog.

24: Math in Macros

At the risk of repeating myself: tables are great things. Not only for ease of otherwise complex formatting, in places where you'd never think of using a table (set borders to no print, and columns and white space in page layout become much easier to handle with precision), but also since tables work so well with macros: there are plenty of macro table commands to work with, so a macro can deal with text or numbers in a table more easily than with data floating loose, so to speak, in your file.

Calculating an increment

Experts that we are, we can go further than that, and turn WP into a spreadsheet. Let's look at a macro that performs a common task when using tables for numerical data: incrementing the value in a column or row of cells. To make this function more useful, we'll design it to calculate the increment for cells the user has selected, rather than necessarily an entire column or row. It works as shown in figure 43: first, the user enters two values in adjacent cells, thereby setting the increment. Then, the user selects the cells that the increment should fill, and runs the macro. The selected cells are filled with the increment. Decimal precision in macros is good to four digits.

You could do all this manually with table formulas, but it would take a lot longer and wouldn't be nearly as much fun to watch. So let's do it better. As a bonus, we'll add code so that if what's in the first cell is text, not a number, the macro will repeat that text throughout the selection. Let's complete the spreadsheet analogy by letting the user put a formula in the first cell and select cells. Our macro will paste that formula with relative addressing throughout the selection!

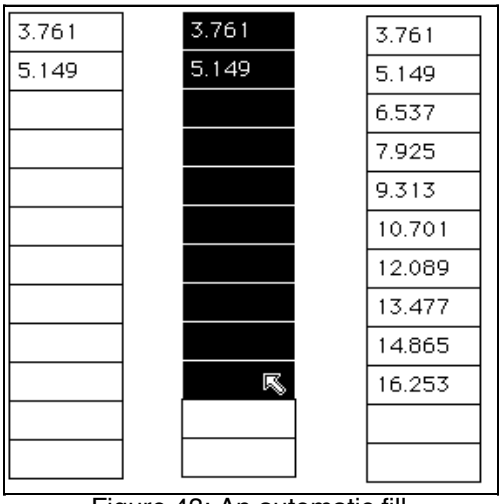


Figure 43: An automatic fill

Listing variables on comment lines

This script starts with a list, on comment lines, of variables used. With this many variables, it's a good idea to start your code with a list, not only so that others can figure out what you did, but so you can figure it out the next time you want to modify the code.

```
; Local variables:
; Var01 = current column number
; Var02 = current row number
; Var03 = ending table column number
; Var04 = ending table row number
; Var05 = string length of clipboard
; Var06 = string length of clipboard minus final return (table cell) code
; Var07 = position of decimal point in numeric string
; Var08 = number of digits
; Var09 = precision of numeric value of first cell
; Var10 = working precision of increment
; Var11 = numeric value of current cell
; Var12 = numeric value of next cell
; Var13 = numeric value of increment
; Var14 = string, at decimal precision of increment
; Var20 = column/row flag
; Var21 = starting column number
; Var22 = starting row number
; Var25 = flag for text or empty cell
```

As you see, a lot of the script will concern itself with data in the existing document, rather than just issuing commands. The latter type of macro is most often less powerful and certainly less sophisticated than the script which looks at the document first, and then acts on it.

Nuts and bolts

Note how this script makes more extensive use of the Call command, to divide the script into a main routine and subroutines. The more complex your macros get, the more helpful this will be. The code is compact, with nested If/Else conditions. Note also the use of text as flags, again in the service of clarity. See how many more indications you can find that my Tech Editor rewrote this one.

The code itself, with comments:

```
; Check for valid selection and mark beginning and ending cells
If (!SelectionFlag!!InTableFlag)
    Alert ("Please select the table cells you want to fill.")
    Go (end)
End If
Assign (Var01;TableColumnNum)  ; the column in which the selection starts
Assign (Var02;TableRowNum)  ; the row in which the selection starts
Down ()  ; deselect all cells, leaving the insertion point in the last cell selected. Just like
pressing the down arrow key in the program.
Assign (Var03;TableColumnNum)  ; ending column of selection
Assign (Var04;TableRowNum)  ; ending row
Assign (Var21;Var01)  ; store the position of the first cell again. We want to return to
Assign (Var22;Var02)  ; that cell at times but will be manipulating var01 or 02
```

```
If (Var01=Var03)
    If (Var02=Var04)
        Alert ("You have selected only one cell.")
        Go (end)
    Else
        Assign (Var20;"column")  ; set flag
    End If
Else
    If (Var02=Var04)  ; same as for row above
        Assign (Var20;"row")
    Else
        Alert ("Your selection should be limited to cells in one column or in one row.")
        Go (end)
    End If
End If
;
; Check the first two cells to find the working precision and the increment, and to see if
either contains text or is empty.
Position To Cell (TableID;Var01;Var02)  ; return to the first cell
Call (getPrecision)  ; get the precision of this cell. If the cell does not contain a number
this won't mean much but the procedure will complete without an error.
Assign (Var10;Var09)  ; store the precision to compare it with the precision of the next
cell
String To Number (Var11;CellValue)  ; store the number in this cell to calculate the
increment
Call (textCheck)  ; check if the cell contains text or is empty
Call (increment)  ; increment either the row or column variable
Position To Cell (TableID;Var01;Var02)  ; go to the next cell
Call (getPrecision)
If (Var10<Var09)
    Assign (Var10;Var09)  ; store the working precision
End If
String To Number (Var12;CellValue)  ; store the number in the second cell
Assign (Var13;Var12-Var11)  ; this is the increment
Call (textCheck)  ; check if the second cell contains text or is empty ;
If (Var25="text")  ; if either cell contains text or is empty
    Assign (Var01;Var21)  ; reset position, counters and clipboard to the first cell
    Assign (Var02;Var22)
    Position To Cell (TableID;Var01;Var02)
    Select TableCell
    Copy
    Down ()  ; can't position to another cell while this cell is still selected, so deselect it
    now
End If
;
; Fill the table cells. At this point, if the first two cells contain numbers we are in the
second cell, if not our position and counters are set to the first cell
While (Var01<Var03|Var02<Var04)  ; if we aren't in the last cell of the selection yet
    Call (increment)
    Position To Cell (TableID;Var01;Var02)  ; go to the next cell
    Select TableCell
    If (Var25="text")
        Paste
    Else
        Clear
        Assign (Var12;Var12+Var13)
```

```
            Number To String (Var14;Var10;Var12)
            Type Var (Var14)
        End If
    End While
    Go (end)
    ;
    Label (getPrecision)
    Assign (Var09;0)
    SubString Position (Var07;".";CellValue)  ; is there a decimal point in the numeric string?
    If (Var07)
        String Length (Var08;CellValue)  ; how many digits?
        Assign (Var09;Var08-Var07)  ; how many digits to the right of the decimal point?
        If (Var09>4)
            Assign (Var09;4);macros are accurate only to decimal precision of 4
        End If
    End If
    Return
    ;
    Label (textCheck)  ; if the cell contains text or is empty, set var25 as a flag
    Select TableCell  ; we'll compare the numerals in the cell to the total contents of the cell
    Copy  ; which we'll do with the clipboard and the cellvalue variables
    Down ()
    String Length (Var05;Clipboard)
    Assign (Var05;Var05-1)
    SubString (Var06;1;Var05;Clipboard)  ; strip the final return character from the clipboard value
    If (Var06!=CellValue|Var06="")  ; if cell contains text or is empty
        Assign (Var25;"text")  ; set flag
    End If
    Return
    ;
    Label (increment)
    If (Var20="column")
        Assign (Var02;Var02+1)
    Else
        Assign (Var01;Var01+1)
    End If
    Return
    ;
    Label (end)
    Position To Cell (TableID;Var21;Var22)  ; return to the first cell
    Calculate Table
    End Macro
```

That's it! Complex overall, maybe, but not difficult when broken down into its basic parts.

I've spent much of this chapter basically commenting code, since looking at samples of code is by far the best way – really, the only way – to learn advanced scripting. Anytime you see a macro whose operation or interface interests you, dissect the code as we've done here. You'll steadily gain in your expertise.

In the last chapter, we'll look at some final touches in elegance and bulletproofing – at which point you'll truly be able to say that you've mastered WordPerfect macros.

25: Bloopers, and Elegance

After the long and complex code discussed in the last few chapters, I'd like to finish this book with some shorter snippets of code that exemplify ways that your scripting can go wrong, and ways it can go right. The first category is largely populated by bugs. Let's look at a couple.

What's wrong here?

A lot of mistakes occur when you script some action and then, later, add code for another feature. The second feature torpedoes the first. In the code below, part of the Tab command for my outlining macros, I wrote the part below the separator first. It checks to see if the next object to the right of the insertion point is a hard return code. I wrote it; it worked; everything's fine. Then, a week later, I was going through all the outlining commands to add checks so things would work as they should. In this case, I decided to check whether the user was in fact in an outline. This code, above the separator, puts the insertion point at the beginning of the current line, and then searches for the next automatic paragraph number (outline label).

At that point, the original code didn't work any more. Not hard to see why, in retrospect. If I send the insertion point looking for the next paragraph number, the object following won't be a hard return code, since my outliner puts an indent immediately after every paragraph number. Hmm. Oops. Duh.

```
Display (Off)
Beginning of Line ()
Find Next Code (Forward;Paragraph Number)
If (!FindStatusFlag)
    Alert ("I can't find a topic to tab.")
    End Macro
End If
; ============================
Raw Read (anything;Right)
If (RawObject=[Return-Hard Code])
    Find Next Code (Backward;Paragraph Number)
    If (FindStatusFlag)
        If (!Var04)
            Assign (Var04;1)
        End If
    End If
End If
```

Smile – it gets worse

Here's an example from my Notes Editor set. This is the part that saves the front window; then, if that document is a footnote or endnote editor, looks for the original paper and saves that too.

If the front window is not an editor but there's an editor open for that paper, the macro again saves both files – the macro essentially handles a paper and a note editor as though it were one document, for the user's convenience. So, while the macro saves editors and other documents with related names, it doesn't touch other open files.

To do this, I look for the substrings "Footnotes" and "Endnotes" in the name of the front window. If I find either, I then take the rest of the name of the front window and look for an open document with that string, now assigned to Var04, as part of its name. If I don't find either, I know that a paper (or any document, with or without an open note editor), is the active document. This sequence gets the name of the paper, wherever it is. Comments follow in the code:

```
Assign (Var01;FrontWindow)
SubString Position (Var02;">~Footnotes";FrontWindow)
SubString Position (Var03;">~Endnotes";FrontWindow)
If (DocumentModifyFlag)
    Save
End If
If (Var02)
    SubString (Var04;2;Var02-2;FrontWindow) ; if there's a Var02, then we get a
    Var04, which will be the paper name, or most of it
End If
If (Var03)
    SubString (Var04;2;Var03-2;FrontWindow) ; or, if there's a Var03, then we get a
    Var04
End If
If (Var04)
    Select Window (Var04)
    If (DocumentModifyFlag)
        Save
    End If
End If
On Error (endnote) ; this starts a sequential set of error handlers
If (!Var02)
    Select Window ("<"$Var04$">~Footnotes") ; if there's not a Var02, then work with
    Var04
    If (DocumentModifyFlag)
        Save
    End If
End If
Label (endnote)
On Error (end)
If (!Var03)
    Select Window ("<"$Var04$">~Endnotes") ; if there's not a Var03, then work with
    Var04
    If (DocumentModifyFlag)
        Save
    End If
End If
```

Thus, if there's a Var02 or Var03, assign a Var04. Then, if there's neither a Var02 nor a Var03, so no Var04, find a window with Var04 as part of its name. Hey, John. Wanna go back to QuicKeys?

This is what can ensue when you stare at the computer for too long at a time. Remember that scripts do exactly what you tell them to do, and what that is can make less sense than you thought. It doesn't mean you're a newbie – Apple Computer's corporate headquarters isn't on Infinite Loop Drive in Cupertino, CA for nothing. It happens. Step through your macro, make a flowchart, or comment out (put semicolons in front of lines) the part of the script that isn't working, and reactivate that code in small parts at a time. In general, looking at the big picture will get you out of the trouble that looking at too many details got you into.

Another good idea is to keep things simple. The example above picks out an endnote editor and a footnote editor for the active document, ignoring other editors as well as other open documents. But who's going to be working with more than one endnote editor at once? Open something to copy a citation, sure, but to save working editor(s) and the paper, the following does fine:

```
On Error (otherEditor)
Select Window ("*Endnotes")
If (DocumentModifyFlag)
     Save
End If
;
Label (otherEditor)
On Error (paper)
Select Window ("*Footnotes")
If (DocumentModifyFlag)
     Save
End If
;
Label (paper)
On Error (end)
If (DocVar7)
     Select Window (DocVar7)
End If
;
Label (end)
If (DocumentModifyFlag)
     Save
End If
```

and in fact, who needs to save editors and paper only, without touching other open documents? So how about:

```
Assign (Var00;FrontWindow)
If (NumberOfWindows>0)
     Repeat
          Cycle Windows
          If (DocumentModifyFlag)
               Save
          End If
     Until (FrontWindow=Var00)
End If
```

rather than the macro equivalent of bloatware. Thoreau said to simplify code whenever possible.

Who's on first?

Another way out of trouble is testing your script at various places. Does that variable still contain what you think it does? Temporarily adding a line like:

```
Alert (Var01)
```

will show you that – as long as Var01 contains a string. The Alert command can't take a number. It can, though, take a number joined to a string– that is, coerced to a string, like:

```
Alert (""$Var01)
```

to help hunt down insects.

Macros à la elegance

Now then. Let me redeem myself, and write some code that runs. In fact, I'd like to finish this book with a return to what most advanced users do with the macro feature most of the time: build something short and sweet that especially fits their working style and needs.

Of course, you can do that already, so I'd like to pay attention here to some fine points of making a script 1) bulletproof, and 2) elegant. These are nice things for a macro to be.

Here's a script I cooked up to access QuickCorrect, a terrific feature, more effectively. In the stock program, you go into a dialog box and take several steps, including retyping the misspelling you want to add to QuickCorrect's word list. This was just enough extra effort that I'd often skip it, and then be stuck making the same typing mistakes over and over. Was there any hope? Yes – a macro!

The one-minute macro

First, the basics. What I wanted to set up was a command that would take and use the misspelling as it originally occurred, right there in the document. A small dialog would then ask the user for a replacement. Armed with those two pieces of data, the macro then adds the word/replacement pair to QuickCorrect's word list *and* makes the correction in the document.

In the online macro help, I saw that the macro **token** (or command) for QuickCorrect's add-a-correction feature was:

```
Add Replacement (Word;Replacement)
```

where both the parameters are character expressions: strings in quotes or variables containing strings. Why not select and copy the misspelling in the document, giving us the first parameter? We could ask the user to type a replacement in a small dialog, and we could utilize that not only as the second parameter for QuickCorrect, but also type it into the document, with the Type Var command, to replace the selected misspelling while we're at it. So we can start with:

```
Select Word
Copy
Get Text (Var01;"Add QuickCorrect Entry";"Enter replacement:")
Add Replacement (Clipboard;Var01)
Type Var (Var01)
```

which takes care of the basics.

Bulletproofing

Often, a script runs fine in the context the programmer envisioned. Users' behavior is hard to predict, though. Our code so far, for example, uses the Select Word command in a blithe assumption that there will in fact be a word to select. What if there isn't, or what if the command selects the wrong word – which the macro will most certainly do if the insertion point isn't where we thought it would be?

This is less of a problem if the user is going to run the macro at the leading edge of the document but, if the user has gone back into an existing paragraph to add some text, there will be text to the right of the insertion point as well as to the left. If the insertion point is pushing existing text along as it goes, and the user has added a space after the misspelling (either inadvertently, or to see whether there's already a QuickCorrect entry for it), the Select Word command will act on the wrong word. If the insertion point is not adjacent to or within a word (defined as some characters with a space or punctuation on both sides), the Select Word command is even less helpful.

The first problem is easy enough to fix. We'll start the macro with the line: **Left ()** to get the insertion point back to the word we want, if we're a space to the right of it. If there's no space, this will move the insertion point into the word, which is fine.

The raw facts

The second potential problem is that the Select Word command might not find anything to select. We can address this with a simple check, using a command similar to Raw Read, which we learned in Chapter 22. That command reads to the left or right of the insertion point, moving the insertion point as it does so. The command we'll use this time, **Raw Look**, does much the same but does not move the insertion point. Both commands put the type of object they find into the RawObjectType variable, also in Chapter 22. Both also put the contents of what they find in the **RawObject** variable, which we'll use this time.

Our strategy would be to see what character is to the immediate left of the insertion point. If it's a space (remember, we've already moved left a space), we aren't close enough to a word for Select Word to work. The code is:

```
Raw Look (char;Left) ; what's the next character to the left?
If (RawObject=' ') ; if a space
    Go (alert) ; tell the user and quit
Else
    Select Word ; we have a word to work with
End If
```

As an alternative, we could use something like:

```
Repeat
    Raw Read (char;Left)
Until (RawObject!=' ')
```

to get to the word (or character) to the left, if there is such a thing and no matter how far. But macros that perform action at a distance can be disconcerting to the user. They're very cool, though.

Anyway, at this point we can use the Copy command, as in the first sample script above.

More fine points

But let's not use Copy right here. That command replaces anything that's already on the clipboard, and the user might have something on the clipboard that's worth keeping, other things being equal. Let's use the Copy command at the last moment possible.

That is going to be a step later than in the basic script shown above. In the meantime, the macro presents a dialog asking for the replacement. At this point, the user could decide not to do anything – the word is correct after all, or it's not a mistyping the user makes often enough to want to add to QuickCorrect, or whatever.

In this case, why grab the clipboard? Let's post the replacement dialog first, and then, if the user actually puts something in that dialog, we can go ahead and copy the text selection.

If the user doesn't put anything in the text entry field of the dialog, he or she would then want to cancel it, by reaching for the mouse and clicking the Cancel button. Let's add another nice touch here: that the user can cancel the dialog by pressing Return – with no text entered. It's just that much faster and easier, which is what we're up to in the first place. This we do with:

```
Get Text (Var01;"Add QuickCorrect Entry";"Enter replacement:")
If (!Var01) ; the dialog was closed and Var01 stayed empty
    Go (cancel)
End If
Copy
```

so the macro branches to a Cancel label if no text was entered, and copies the selection to the clipboard otherwise.

A Cancel label gives us another option too. Without it, if the user clicks the Cancel button in the dialog, the macro ends, leaving a word selected. No real problem, but I try to write macros that do not leave selections behind them. This is simply because if the user isn't watching things, his or her next action might be to type something, thereby replacing the selection. If what was originally selected turns out to be something the user wants to keep, and that word then disappears in the middle of some fast-paced word processing, let's hope the user is lucky enough to find it later when proofreading. Better yet, let's give this a simple fix with a Cancel label at the end of the macro with only two lines:

```
Label (cancel)
Right ()
```

deselecting the word, and putting the insertion point to its immediate right, just where the user is going to want it.

And let's start the whole macro with an On Cancel handler:

```
On Cancel (cancel)
```

which will direct execution to the Cancel label if the user clicks the Cancel button *or* presses Return with an empty dialog. So no matter what the user does, the macro won't leave selected text.

Is all this "good design" worth it? I sure think so. Consider that it takes a minute or two to polish your script, and your users will appreciate those extra steps – especially if it means they won't lose work.

And we end up with this:

```
On Cancel (cancel)
Left ()
Raw Look (char;Left)
If (RawObject=' ')
    Go (alert)
Else
    Select Word
End If
Get Text (Var01;"Add QuickCorrect Entry";"Enter replacement:")
If (!Var01)
    Go (cancel)
End If
Copy
Add Replacement (Clipboard;Var01)
Type Var (Var01)
End Macro
Label (alert)
Alert ("Your insertion point must be within a word, or only one space to the right of a
word.")
End Macro
Label (cancel)
Right ()
```

Looking forward . . .

Not in the sense of raw look, but continuing to learn WP macros – I've said it before and, if this weren't the last chapter I'd say it again: analyzing scripts is by far the best way to make progress. Any time you see a macro whose operation or interface interests you, dig into the code, and maybe flowchart the structure, until you see why the author did it that way. You'll steadily gain in your expertise.

This concludes my tour of WordPerfect's macro language. You're now a master of the most advanced feature of the best word processing program on the best computer there is, to make your writing and editing efforts far more creative and productive than they'd otherwise be. Thanks for joining me, and have fun!

Appendix A: Error Messages

Messages within the macro editor window

These appear as a single line at the bottom of the script window, and refer to underlined code in your script. If you attempt to save, you'll get an error dialog like figure 7 on page 7.

Error: Command Parameters Are Too Long	Parameter length in the macro editor is 512 bytes (about 255 characters). This should be a problem only with a text parameter, or a long list of variables.Try putting text in variables before the problematic command, then using the variables in the command itself. Or, join variables first.
Error: Expected a Number, Variable, or Numeric Expression	A string is one type of data; a number is another. You may have one type in a variable or parameter where the macro editor cannot make sense of it.
Error: Expected a String, Variable, or Alphanumeric Expression	A string is one type of data; a number is another. You may have one type in a variable or parameter where the macro editor cannot make sense of it.
Error: Not a Legal Expression	Check spelling and punctuation of your syntax in the online help. Be especially careful with parentheses, curly brackets, and spaces.
Error: Parameter Is Too Long	Command length in the macro editor is 512 bytes (about 255 characters). Only the excess length is underlined.Try using one or more variables to contain the text in the problematic parameter or, if there are too many variables, join a few first.
Error: This Type Not Allowed Here	Bad syntax, of almost any type not addressed more specifically in another error message.Check your spelling against the online help; pay particular attention to spaces and punctuation.
Error: Too Many Parameters	Check the command syntax against the on-line help. This is a common error when using similar commands: for example, Get Text doesn't take a maximum length parameter, while Get String does.
Error: Unexpected Characters After ')'	A common mistake when reading a text file into the macro editor. There is a space or something else, possibly invisible, following the close parenthesis. Generally, the close parenthesis comes at the end of a command line.
Error: Unexpected End Of Line	You may have pressed Return at the wrong place. In any case, something is missing from the underlined command line. This could easily be a closing parenthesis
Error: Unrecognized Command	Check the spelling of the underlined command against the on-line macro help.
Error: Unrecognized Parameter Name	Check the spelling of the underlined parameter against the on-line macro help.

Error: Unrecognized Read/Write Variable Name	Check the syntax and spelling of the underlined variable against the on-line macro help.
Error: Unrecognized Variable Name	Check the syntax and spelling of the underlined variable against the on-line macro help.

Dialogs indicating errors in reference

The following messages appear as alerts when you attempt to run a macro with correct syntax, but incorrect flow structure or data types.

Macro Terminated: Could not chain or run the macro specified.	You have misspelled the name of another macro, or that macro isn't installed.
Macro Terminated: Could not find corresponding ELSE, END IF, END WHILE or END FOR.	Flow structure incomplete. With automatic indentation in the editor, this is usually easy to find.
Macro Terminated: END FOR encountered without corresponding FOR or FOR EACH.	Flow structure incomplete. With automatic indentation in the editor, this is usually easy to find.
Macro Terminated: END WHILE encountered without corresponding WHILE.	Flow structure incomplete. With automatic indentation in the editor, this is usually easy to find.
Macro Terminated: Error evaluating an expression for this command.	Mixed data types. You may be treating a variable containing a string as though it contained a number, for example.
Macro Terminated: Error in UNTIL expression.	Invalid syntax or flow structure. Check syntax in the online help.
Macro Terminated: Error in WHILE expression.	Invalid syntax or flow structure. Check syntax in the online help.
Macro Terminated: Error performing operation while evaluating an expression.	Mixed data types. You may have asked the macro to add a string to a number, for example.
Macro Terminated: Error reading parameter for this command.	A command parsed right but still won't run. Turn Step on to identify the offending line, and check its syntax. Check values and data types of variables.
Macro Terminated: Error returned by nested macro or macro subroutine.	The subroutine or nested script you called has either encountered an error during its own execution, or has returned data that the parent macro can't work with.
Macro Terminated: Expected a number, but got something else.	Mixed data types. Make sure whether you're dealing with a string or number.
Macro Terminated: Expected a numeric string but got something else.	Mixed data types. Note: a numeric string is tricky. It's legally a string, but consists only of numerals, and perhaps a decimal point.
Macro Terminated: Expected Alphanumeric string but got something else.	Mixed data types. Make sure whether you're dealing with a string or number.

Macro Terminated: FOR encountered but END FOR could not be found.	Invalid syntax or flow structure. Check syntax in the online help. Automatic indentation in the editor may make the problem easier to find.
Macro Terminated: No expression found to assign in FOR EACH.	Check syntax in on-line help.
Macro Terminated: Out of Memory.	Quit WP and increase RAM allocation in the Finder.
Macro Terminated: Specified LABEL could not be found.	A label in the flow structure is misspelled or nonexistent.
Macro Terminated: Specified window could not be found	The macro is looking for a window that isn't open. This is a good place for an error handler.
Macro Terminated: This command not valid here.	You may have a Copy command when there's no text selected, for example.
Macro Terminated: This is not a valid macro command.	Check spelling of syntax n the on-line help.

Appendix B: Read-Only Variables

Alignment
AOCEFlag
AutoHyphenation
BackBlue
BackGreen
BackRed
BackWindow
BoldFlag
BookMarkBarFlag
BootDir
BottomMargin
ButtonBarFlag
ButtonBarPosition
CellValue
ChapterNumber
Clipboard
ColorQDFlag
ColumnMode
ColumnNumber
CurrentAscent
CurrentDescent
CurrentDir
CurrentLeading
CurrentStyle
CursorDocV
CursorH
CursorPageV
DecimalPointChar
DocLanguageID
DocRegionID
DocScriptID
DocumentModifyFlag
DocumentName
DoubleUnderlineFlag
EndnoteBarFlag
EndnoteNumber
ExtraLargeFontFlag
FillBackgroundColorBlue
FillBackgroundColorGreen
FillBackgroundColorRed
FillForegroundColorBlue
FillForegroundColorGreen
FillForegroundColorRed
FillPattern
FindDirection
FindStatusFlag
FineFontFlag
FontBarFlag
FontName
FontScriptID
FontSize
FooterBarFlag
FootnoteBarFlag
FootnoteNumber
ForeBlue
ForeGreen
ForeRed
FormatOrientation
FrontWindow
GraphicBoxNumber
GridSizeHorizontal
GridSizeVertical
HeaderBarFlag
HiliteBlue
HiliteGreen
HiliteRed
HTMLBarFlag
InDrawFlag
InTableFlag
InTextBoxFlag
ItalicsFlag
KeyScriptID
LargeFontFlag
LayoutBarFlag
LeftCellMargin
LeftColumnMargin
LeftMargin
LineCharacterCount
LineNumberingFlag
LineSize
LineSpacing
ListBarFlag
LogicalLine
LogicalPage
MailerBarFlag
MathBarFlag
Measurement
MergeBarFlag
MovieCount
MultipleScriptFlag
NewDocumentFlag
NextWindow
NumberOfColumns
NumberOfObjects
NumberOfWindows
ObjectID
ObjectType
OutlineFlag
PageHeight
PageNumberPosition
PageNumberType
PageWidth
PCFormatFlag
PenBackgroundColorBlue
PenBackgroundColorGreen
PenBackgroundColorRed
PenForegroundColorBlue
PenForegroundColorGreen
PenForegroundColorRed
PenPattern
PhysicalLine
PhysicalPage
PICityState
PIFax
PIName
PIOrganization
PIPhone
PIPosition
PIStreet
PlainFlag
Random
RawObject
RawObjectType
ReadOnlyDocumentFlag
RedlineFlag
ReplacementWord
RightCellMargin
RightColumnMargin
RightMargin
RoundedRectSizeHorizontal
RoundedRectSizeVertical
RulerBarFlag
RulerFlag
SaveAsFormat
ScaleFactor
ScreenSizeH
ScreenSizeV
ScriptVarCount
SelectionFlag
ShadowFlag
ShowCodesFlag
ShowScriptFlag
ShowWhiteSpaceFlag
SmallCapsFlag
SmallFontFlag
SmartQuotesFlag
SnapToRulerFlag
SpacingBelowColumns
SpacingBetweenColumns
SpeechBarFlag
StatusBarFlag
StrikeoutFlag
StylesBarFlag
SubscriptFlag
SuperscriptFlag
SystemLanguage
TabAlignChar
TableBarFlag
TableBoxNumber
TableColumnNum
TableID
TableMaxColumnNum

TableMaxRowNum
TableRowNum
TextBlue
TextBoxNumber
TextGreen
TextRed
ThousandsSep
TOABarFlag
TopMargin
UnderlineFlag
UserBoxNumber
VeryLargeFontFlag
WidowOrphanFlag
WordToReplace
WPDir
WPLibraryName

Appendix C: Code Values

Align Center	-12282
Align Justify	-12282
Align Justify All	-12282
Align Left	-12282
Align Right	-12282
Alignment Character	-11520
Back Tab	-16125
Block Protect Off	-15103
Block Protect On	-15104
Bold Off	-15616
Bold On	-15616
Book Mark	-9982
Border-Character Off	-8448
Border-Character On	-8448
Border-Column Off	-8446
Border-Column On	-8446
Border-Page Off	-8445
Border-Page On	-8445
Border-Paragraph Off	-8447
Border-Paragraph On	-8447
Box Number	-25600
Center Line	-16127
Center Page Top to Bottom	-22528
Chapter Number	-25344
Chapter Number Options	-11765
Chapter Number Set	-11502
Color Print-Text	-12032
Column Break-Hard	-9208
Column Definition	-11775
Columns-Vertical Space Between	-12278
Date/Time Code	-10240
Double Underline Off	-15604
Double Underline On	-15604
End of Generated Text	-22784
Endnote	-10751
Endnote Number	-23552
Endnote Numbering Options	-11517
Endnote Options	-11772
Equation Box	-9724
Extra Large Font Off	-15603
Extra Large Font On	-15603
Figure Box	-9728
Figure Box Numbering Options	-11508
Figure Box Options	-11771
Fine Font Off	-15599
Fine Font On	-15599
Flush Right	-16126
Font Change	-12031
Font Size	-12030
Footer A	-11006
Footer B	-11005
Footnote	-10752
Footnote Number	-24064
Footnote Numbering Options	-11518
Footnote Options	-11773
Frame-Equation Off	-8440
Frame-Equation On	-8441
Frame-Figure Box Off	-8444
Frame-Figure Box On	-8444
Frame-Text Box Off	-8442
Frame-Text Box On	-8442
Header A	-11008
Header B	-11007
Horizontal Line Code	-9722
HTML - Heading 1	-9976
HTML - Heading 2	-9976
HTML - Heading 3	-9976
HTML - Heading 4	-9976
HTML - Heading 5	-9976
HTML - Heading 6	-9976
HTML Beginning of Tag	-15300
HTML End of Tag	-15298
HTML Escape	-15322
HTML Image Box	-9723
HTML – Address	-9975
HTML – Blink	-9975
HTML – Cited work	-9975
HTML – Emphasis	-9975
HTML – Keyboard	-9975
HTML – Preformatted	-9975
HTML – Sample	-9975
HTML – Script (Java)	-9975
HTML – Source code	-9975
HTML – Strong	-9975
HTML – Typewriter	-9975
HTML – Variable	-9975
Hyper Link	-9981
Hyphen-Automatic	-9200
Hyphen-Required	-27136
Hyphen-Soft	-26880
Hyphenation Off	-19200
Hyphenation On	-19456
Hyphenation Zone	-12285
Hyphenation-Suppress	-26112
Indent-First Line	-12276
Indent-Left	-15872
Indent-Left/Right	-15871
Index Generated Here	-10494
Italics Off	-15615
Italics On	-15615
Keep Together Next 'N' Lines	-9983
Kerning	-16124
Language	-11503
Large Font Off	-15601
Large Font On	-15601
Leading	-12029
Line Height	-12288
Line Numbering Font/Size	-11764

Line Numbering Options . . . . . -11515
List Generated Here . . . . . -10494
Margin Set-Left/Right . . . . . -12287
Margin Set-Top/Bottom . . . . . -12283
Mark Index . . . . . -10493
Mark List . . . . . -10484
Mark Reference-Define Target . . . . . -10488
Mark Reference-Refer to Target . . . . -10489
Mark Table of Authorities . . . . . -10492
Mark Table of Contents . . . . . -10496
Merge-Chain to Macro . . . . . -8703
Merge-Data File . . . . . -8703
Merge-Data File Message . . . . . -8703
Merge-Date . . . . . -8704
Merge-Define Names . . . . . -8704
Merge-End of Field . . . . . -8704
Merge-End of Record . . . . . -8704
Merge-Field Name . . . . . -8703
Merge-Field Number . . . . . -8703
Merge-Field Number Message . . . . . -8703
Merge-Field Prompt . . . . . -8703
Merge-Form File . . . . . -8703
Merge-Form File Message . . . . . -8703
Merge-From Keyboard . . . . . -8704
Merge-Keyboard Response . . . . . -8703
Merge-Macro File Message . . . . . -8703
Merge-Next Record . . . . . -8704
Merge-Notification Message . . . . . -8703
Merge-Quit . . . . . -8704
Merge-To Printer . . . . . -8704
Merge-Transfer . . . . . -8704
Misspelled word - Begin . . . . . -15587
Misspelled word - End . . . . . -15587
Outline Off . . . . . -15612
Outline On . . . . . -15612
Overstrike . . . . . -10238
Page Break-Hard . . . . . -9209
Page Break-Soft . . . . . -9215
Page Break-Soft/Hard Return . . . . . -9213
Page Number . . . . . -24320
Page Number Position . . . . . -12280
Page Numbering Font/Size . . . . . -11763
Page Numbering Odd/Even . . . . . -11513
Page Numbering Options . . . . . -11516
Paragraph Number . . . . . -10239
Paragraph Numbering Options . . . . . -11774
Published Edition . . . . . -7936
Redlining Off . . . . . -15608
Redlining On . . . . . -15608
Relative Font Size Definition . . . . . -12028
Return-Dormant . . . . . -9210
Return-Hard . . . . . -9214
Return-Soft . . . . . -9216
Shadow Off . . . . . -15611
Shadow On . . . . . -15611
Small Caps Off . . . . . -15598
Small Caps On . . . . . -15598
Small Font Off . . . . . -15600
Small Font On . . . . . -15600
Space-Non-Breaking . . . . . -24576
Space-Required . . . . . -24385
Spacing Between Lines . . . . . -12286
Spacing Between Paragraphs . . . . . -12279
Strikeout Off . . . . . -15607
Strikeout On . . . . . -15607
Style Off . . . . . -9470
Style On . . . . . -9471
Subscribed Edition . . . . . -7933
Subscript Off . . . . . -15606
Subscript On . . . . . -15606
Subtitle-Face . . . . . -8189
Subtitle-Font . . . . . -8191
Subtitle-Position . . . . . -8188
Subtitle-Size . . . . . -8190
Subtitle-Start of Range . . . . . -23296
Subtitle-Text . . . . . -8192
Super/Subscript Options . . . . . -11766
Superscript Off . . . . . -15605
Superscript On . . . . . -15605
Suppress Page Format . . . . . -12281
Tab . . . . . -16128
Tab Align . . . . . -16122
Tab Set . . . . . -12284
Table Definition . . . . . -7679
Table of Authority Generated Here . . . -10494
Table of Contents Generated Here . . -10494
Table-Cell . . . . . -9194
Table-Cell Alignment . . . . . -7667
Table-Cell Attributes . . . . . -7668
Table-Cell Border Bottom . . . . . -7675
Table-Cell Border Color Bottom . . . . . -7671
Table-Cell Border Color Left . . . . . -7672
Table-Cell Border Color Right . . . . . -7670
Table-Cell Border Color Top . . . . . -7673
Table-Cell Border Left . . . . . -7676
Table-Cell Border Right . . . . . -7674
Table-Cell Border Top . . . . . -7677
Table-Cell Fill Color and Pattern . . . . . -7669
Table-Cell Locked . . . . . -7666
Table-Decimal Align . . . . . -12282
Table-End of Cell . . . . . -9194
Table-End of Row . . . . . -9192
Table-End of Table . . . . . -9190
Table-Join Cells . . . . . -7678
Table-Math Formula . . . . . -7424
Table-Math Grand Total . . . . . -7421
Table-Math Number Format . . . . . -7420
Table-Math Sub-total . . . . . -7423
Table-Math Total . . . . . -7422
Table-Row . . . . . -9192
Table-Row Height . . . . . -7665
Text Box . . . . . -9726

Text Box Numbering Options	-11506
Text Box Options	-11769
Underline Off	-15614
Underline On	-15614
Underline Options	-11519
Very Large Font Off	-15602
Very Large Font On	-15602
Watermark A	-11004
Watermark B	-11003
Widow/Orphan Off	-20224
Widow/Orphan On	-20480

Appendix D: My WordPerfect Scripts

Everything's free, at: ftp://ftp.corel.com//pub/WordPerfect/WPMac/Macros/ and http://hyperarchive.lcs.mit.edu/HyperArchive/Abstracts/text/wp/HyperArchive.html.

- Auto Character Styles 1.0.2 provide character-level styles which automatically update, like the program's paragraph styles.
- Bookmark 1.1 macros provide five bookmarks per document, with a menu showing the page number and text selected when the bookmark was set. For program versions 3.0 and 3.1.
- Character Styles 1.1.2 provide character-level formatting styles which complement the program's paragraph styles. "Incredibly useful" – Umich Mac Archive
- Citations 2.1 make WP and any database into a seamless reference manager like EndNote or ProCite, working within WP as Endnote works within Word.
- Custom Prefs let you switch among any number of customized program preferences settings.
- Document Notes add a message to any WP file. Can be set to appear automatically when you open that file. Easily edited, deleted and searched, and set to disappear automatically.
- Equation Manager automatically numbers and formats equations in documents, supports cross-referencing and lists of equations by number and name.
- File Manager 2.2.3 includes batch macro processing: global find/change for all files in a folder; a menu of frequently (not necessarily recently) opened files; worksets of files that can be opened all at once; save and/or print all open documents; and macros to change a set of file or folder names according to the date or day of the week.
- Find Manager 1.0.2 includes QuickFind – a faster find dialog, Find in All Open, and Find Recent, which lets you choose from the five most recent find strings you've found.
- Glossary 1.3 offers up to 26 glossary entries. A menu most of the width of your screen shows you as much of each entry as will fit. Click on the entry or type its identifying letter to place the entry in your document. Or, you can assign keystrokes to individual entries.
- GREP 1.1.2 adds powerful, regular expression search capability - find patterns of text.
- Note Editor 4.0 shows all footnotes or endnotes in one window, automates the cross referencing of notes, and greatly facilitates formatting and printing.
- Outlining Module, now shipping with program version 3.5e (no longer available as macros). Widely regarded as the best outliner available on the Macintosh.
- Previous Positions 2.0 let you remember up to ten places among all open documents, and go to any one with a keystroke
- QuickCorrect 1.2.1: a much faster way to add QuickCorrect entries, correcting the initial mistyping at the same time, and toggle QuickCorrect on and off quickly.
- Table Manager 1.1: automatically fills selected cells with an increment defined by the first two cells of the selection, shades alternate columns or rows of a table, selects parts of a table, navigates numerically to a table cell, and creates a text box with a table in it.
- Thousand Clipboards offers that many (or more) clips for text.
- Tip 'o the Day 1.1 chooses at random from 98 user-definable tips, optionally at startup.
- Tips & Macros 1.6, 50 pages on almost everything to do with WP.
- UltraClip 1.0.7 gives you up to 50 clipboards for text. Clips are named automatically (or you can name them yourself) and saved to disk. Paste by choosing a clip from a menu.
- Window Manager 2.0.1 offers control over all open windows, tiles open windows horizontally (so you can read entire lines of text), toggles first and second windows, and opens a new independently scrollable view of the active document. You can also close, save or print all open documents with one command.

Appendix E: How to Install Macros

1. Open the file containing the macros you want to install. It must be the front window.

2. From the **Edit** menu, choose **Preferences**. You'll see figure 44:

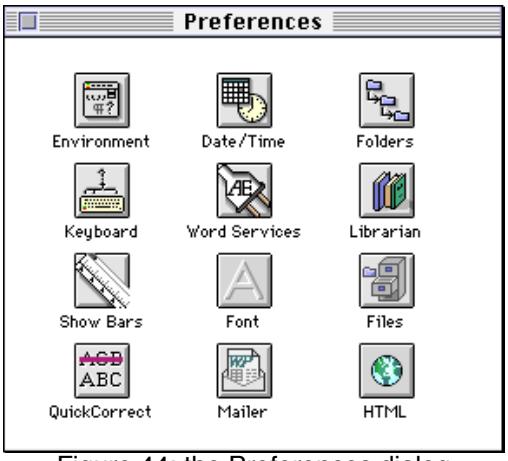


Figure 44: the Preferences dialog

3. Click on **Librarian**. With a front window is titled "Outline Macros and Styles." You'll see figure 45:

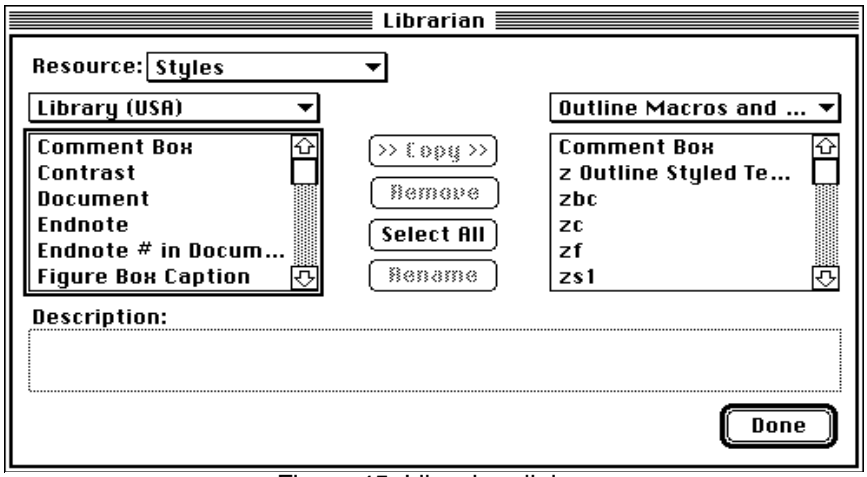


Figure 45: Librarian dialog

4. Click and hold on the **Resource** menu at the top left, which is now set to **Styles**. That menu will then look like figure 46:

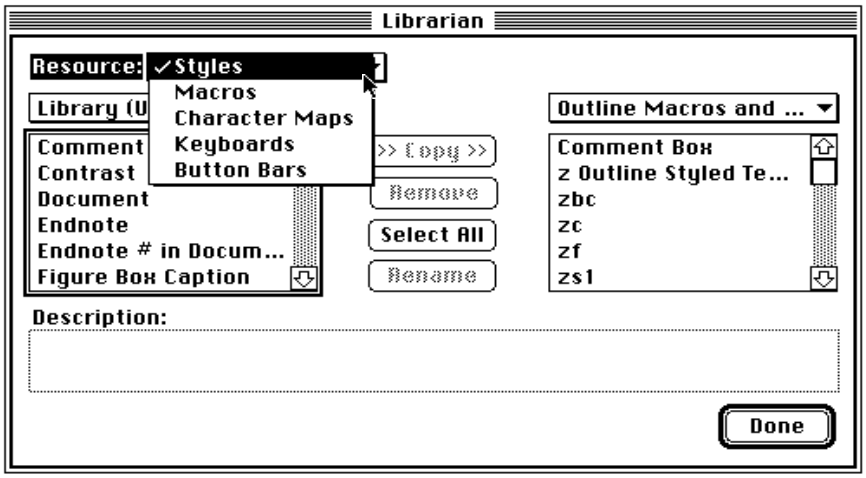


Figure 46: Changing resources in the Librarian

5. Drag until the next word on the menu, **Macros**, is selected, and release the mouse.

6. Click on the list in the box at the right, underneath the title of the front window (in this case, "Outline Macros and …").

7. Click on the **Select All** button in the middle of the dialog. The whole list on the right should become selected, looking like figure 47:

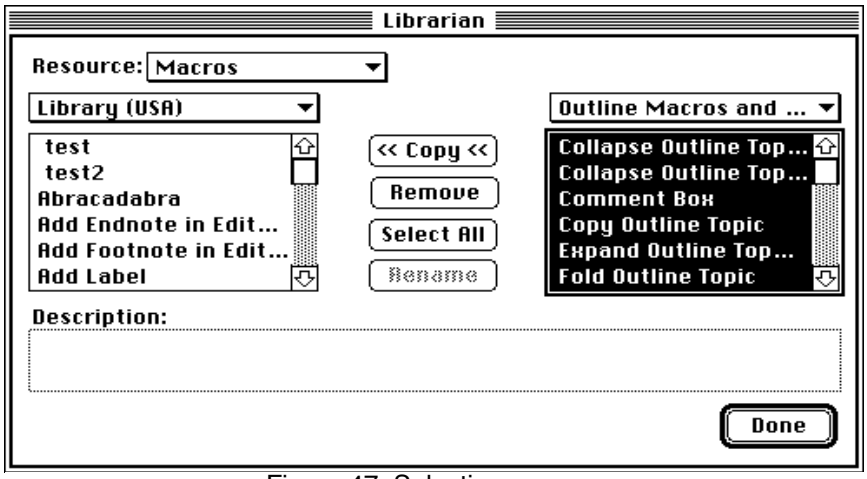


Figure 47: Selecting macros

8. Click on the **Copy** button in the middle of the dialog.

9. After WordPerfect's globe cursor stops spinning, click **Done**.

All of the macros in the front window are now installed in your library, where they're available for use without your having to open these dialogs again.

10. Click in the Close Box of the Preferences window if you don't want to assign any keystrokes to macros. If you do, here's how:

11. From the Preferences window, click **Keyboard**. Then click and hold on the **Type** menu at the top left. You'll see figure 48:

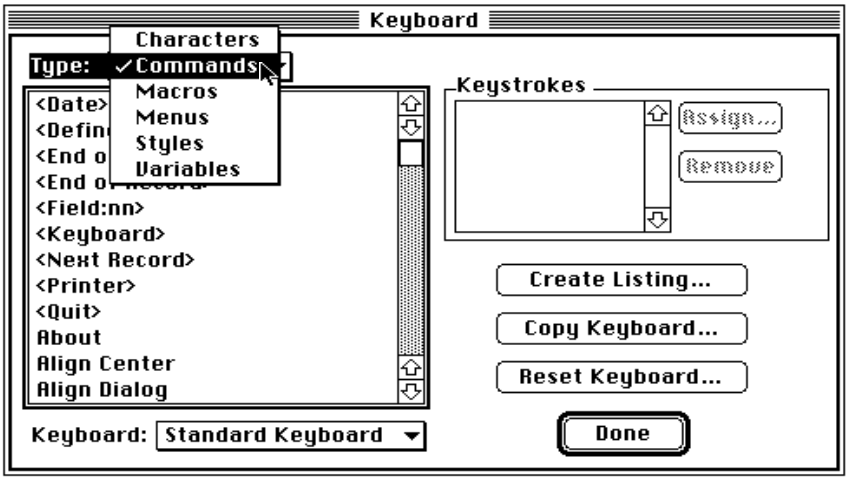


Figure 48: Changing Type in Keyboard

12. Drag until the word **Macros** is selected, and release the mouse button.

13. Using the scroll bar on the list on the left, scroll down until you see the macro you want to run with a keystroke. In this example, the macro is **Comment Box**, and we want to assign it the keystroke **Option-Return**. The dialog will now look like figure 49:

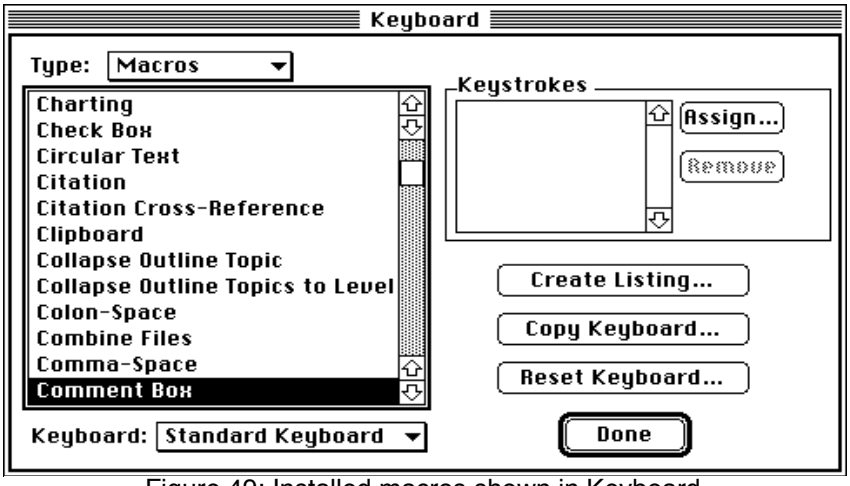


Figure 49: Installed macros shown in Keyboard

14. Click the **Assign** button at the top right. You'll see figure 50:

Figure 50: Assigning a keystroke

15. Hold down the **Option** key on your keyboard, and press **Return**. A graphic representation of that keystroke will appear in the selected box, to confirm your choice. Click **Assign**.

16. Repeat steps #13-15 for any other macros you want to assign keystrokes.

17. Click **Done** in the Keyboard dialog.

18. Click in the Close Box to close the Preferences window.

And that's it!

Index