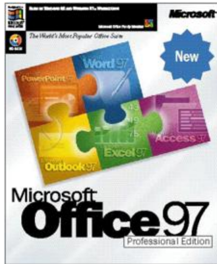
Microsoft Office®

White Paper

Microsoft Word

Published: February 13, 1997
For the latest information, see http://www.microsoft.com/officedev/



Everything for Building and Distributing Solutions with Microsoft Office 97

Converting WordPerfect Macros to Microsoft Word, Visual Basic for Applications

Table of Contents

Preface 2
Resources 2
Supported Versions 3
Introduction 3
The Conversion Process 4
Step 1: Examining the Purpose of the Macro 4
Step 2: Determine if a Macro Is Needed 4
Step 3: Chart the Flow of the Macro 4
Step 4: Record Duplicate Macros in Word 4
Step 5: Modify the Recorded Macros 5
Step 6: Test the Finished Product 5
How Macros Differ Between WordPerfect and Word 5
Deciding When a Macro Isn't Needed 6
Understanding Where Macros Are Stored In Word 7
The Visual Basic Editor: A Basic Roadmap 7
Understanding the Terminology of Visual Basic 8
Comparing Syntax 9
General 10
Recording Macros to Learn Syntax 13
Converting Programming Commands 14
WordPerfect for DOS to Visual Basic Command Cross-Reference 14
WordPerfect for Windows to Visual Basic Command Cross-Reference 16
Converting Variable Assignments 18
Converting Expressions 19
Converting Macros that Insert and Format Text 19
Converting Macros that Use Documents 20
Converting User Input 22
Converting Macros that Pause 23
Converting Alerts 23

Converting Dialog Boxes and Menu Lists 23
Converting Yes/No Messages 25
Converting DLL Calls 25
Converting Arrays 26
Improving upon WordPerfect Macros 27
Additional String Functions in Visual Basic 27
Additional Math Functions in Visual Basic 28
Registry Statements 28
File Functions 29
Obtaining and Setting Current Values from Word 29
Communication with Other Applications 29

Preface

This document describes the methodology, approach, and requirements for converting macros developed for various versions of WordPerfect to Visual Basic® for Applications in Microsoft® Word. Because of the complexity of macros, there's no automatic conversion utility available for converting macros from WordPerfect to Visual Basic for Applications. This document will help you understand the differences between the programming languages used by WordPerfect and Word, and how to successfully re-record or rewrite your WordPerfect macros for use with Word.

Some basic familiarity with both WordPerfect and Word is assumed. This document doesn’t describe the principles of programming for either WordPerfect or Word. If you aren't familiar with the Word or WordPerfect programming languages, you should first learn basic concepts of each language before attempting to convert your WordPerfect macros.

This document provides only a rudimentary discussion of programming using Visual Basic. You may want to augment your study of Visual Basic using Microsoft Office 97 Help, as well as the suggested resources listed under “Resources” later in this paper.

Resources

A valuable resource for general information about switching from WordPerfect to Word 97 is the *Microsoft Office 97 Resource Kit* (http://www.microsoft.com/office/ork/). The *Office Resource Kit* is available in print wherever computer books are sold or from the Microsoft Press® Web site (http://www.microsoft.com/mspress/). The *Office Resource Kit* is also available in online form on the World Wide Web at http://www.microsoft.com/office/ork/.

Also refer to the *Microsoft Office 97/Visual Basic Programmer's Guide*, a comprehensive guide and reference to programming Word and other Microsoft Office applications. The *Microsoft Office 97/Visual Basic Programmer's Guide* (ISBN 1-57231-340-4) is available wherever computer books are sold or from the Microsoft Press® Web site (http://www.microsoft.com/mspress/). It is also available in online form on the World Wide Web at http://www.microsoft.com/officedev/docs/opg/.

For additional resources, visit the Office Developer Forum Web Site (http://www.microsoft.com/officedev/). This site has technical information, code samples, and pointers to additional resources including many books on developing with Microsoft Office 97 and Visual Basic.

Supported Versions

WordPerfect's macro language differs from version to version of WordPerfect. This document describes converting macros from two WordPerfect versions: WordPerfect 5.1 for DOS, and WordPerfect 6.1 for Windows®. If your WordPerfect macros are from a different version of WordPerfect, you may still be able to convert them, but there may be some issues to consider.

- The macro language in WordPerfect 5.0 for DOS is identical to that in WordPerfect 5.1 for DOS, except it contains fewer commands. For all practical purposes, a WordPerfect 5.0 for DOS macro can be considered the same as a WordPerfect 5.1 for DOS macro.
- Versions prior to WordPerfect 5.0 for DOS lacked a programming language, and are therefore not a topic of interest in this document.
- WordPerfect versions 6.0 and 6.1 for DOS use a different macro language than WordPerfect 5.1 for DOS and WordPerfect for Windows. Conversion from WordPerfect 6.0 and 6.1 for DOS are not detailed in this document.
- WordPerfect versions 6.0, 6.1, and 7.0 for Windows are identical except for slight variations in syntax and command availability. For the purposes of this document, they are considered to be the same version.
- WordPerfect 5.1 and 5.2 for Windows uses a now-abandoned macro programming dialect. While the language is similar in some respects to the macro language of WordPerfect 6.1 for Windows, there are substantial differences in the command syntax. For this reason, and since these versions haven't been commercially sold for several years, converting macros from WordPerfect 5.*x* for Windows is not detailed in this document.

Introduction

Macros are commonly used for automating office tasks – using desktop applications to do more in less time. Typical automated office tasks are inserting common blocks of text into documents, formatting documents in a specific style, and automatically assembling larger documents from smaller documents.

Unlike documents, which can often be converted between different word processors using a conversion "filter," macros are really miniature programs and cannot be readily converted. If you have macros that you've recorded or written for WordPerfect, they'll need to be re-recorded or rewritten for use with Word. The difficulty in re-creating automated WordPerfect office tasks in Word depends on the complexity of the original macros. Simple macros are easy to re-create in Word.

Word 97 includes Visual Basic 5.0, a sophisticated development environment that is shared across Office applications: Word, Microsoft Excel, Microsoft PowerPoint®, and Microsoft Access. Visual Basic is also part of the Microsoft Visual Basic product and Microsoft Project. Visual Basic is also licensed to other software companies. Over 40 companies, including Adobe, Autodesk, SAP, and Visio, have announced that they will include Visual Basic in their applications. Visual Basic goes beyond being merely a macro language – it is a full-featured programming development environment. Throughout this document, we'll refer to "macros" in Word as Visual Basic code.

Documentation support for Visual Basic is enormous, with over 50 books printed by Microsoft Press and other publishers. These books run the gamut from tutorial guides for beginners to advanced references for programming

professionals. For information about Microsoft Press titles, see the Microsoft Press Web site at http://www.microsoft.com/mspress/.

The Conversion Process

Converting macros from WordPerfect to Visual Basic is a six step process.

1. Examine the purpose of the WordPerfect macro.
2. Determine if a macro is needed, or if Word can handle the job using another built-in feature, such as AutoText or forms.
3. Chart the flow of the macro to define its important routines.
4. In Word, record one or more macros that duplicate the functionality of the original macro.
5. If necessary, modify the recorded macros and manually add additional programming instructions where necessary.
6. Test the finished Visual Basic code.

Step 1: Examining the Purpose of the Macro

You must fully understand the purpose of the WordPerfect macro before it can be successfully converted. If possible, run the macro on a copy of WordPerfect or view the macro in WordPerfect's macro editor. This will give you a better understanding of what the macro does.

Take note of any documents that the macro uses or produces. Word versions of these documents maybe needed when the macro is converted to Visual Basic.

Step 2: Determine if a Macro Is Needed

Sometime it's not necessary to replicate a WordPerfect macro in Word, especially if the macro performs a rudimentary formatting function, such as applying bold and underlining to text (this task can be easily handled using Word styles). The Word interface, as well as other features, make many of these simple macros unnecessary.

See the "Deciding When a Macro Isn't Needed" section later in this document for more information.

Step 3: Chart the Flow of the Macro

Simple macros do a specific job and nothing more. Complex macros may perform several tasks in a particular order depending on external conditions. If a macro performs a number of steps during its execution, create a simple flowchart that outlines each step. Be sure to include any pauses in the macro for user input, such as answering Yes/No questions or typing text. The instructions that create these pauses need to be manually added to the recorded Word macro.

Step 4: Record Duplicate Macros in Word

There are hundreds of properties, methods and objects available in Visual Basic. Learning them all is a daunting task. The most time efficient approach to macro conversion is to record duplicates of your WordPerfect macros using Word's macro recorder. You can then view the resulting Visual Basic instructions which can then be assembled into larger macros using copy and paste.

Step 5: Modify the Recorded Macros

You can use your recorded Word macro as-is or use the code to build larger macros. You may need to record short segments that duplicate the original functionality of the WordPerfect macro, and then combine these instructions with additional instructions you manually add to build the finished macro. For example, you may add instructions that prompt a user for input, such as asking if the user wishes to perform a certain task.

For more information on recording macros in Word, see "Revising recorded Visual Basic macros" in Word Visual Basic Help (use the **Find** tab to locate the topic).

Step 6: Test the Finished Product

Test your new Visual Basic code to make sure it duplicates the functionality of the original WordPerfect macro. If the WordPerfect macro created a document or some other output, compare the output generated by Word with the output generated by WordPerfect.

Note While you're converting macros to Visual Basic, look for ways to make them better. This is especially valuable when converting WordPerfect for DOS macros. WordPerfect for DOS imposed a number of restrictions on macro programmers, such as limited access to disk and file services, only two documents open at once, and no built-in user interface tools like message boxes. Visual Basic doesn't have the same limitations. Before converting your WordPerfect macros, consider adopting the new features and functionality available in Visual Basic.

See the "Improving upon WordPerfect Macros" section later in this document, for more information.

How Macros Differ Between WordPerfect and Word

Beyond the differences in the programming languages used by WordPerfect and Word, the approach to writing and developing macros varies considerably between the two products.

In WordPerfect 5.1 for DOS, macros are written using a small and somewhat limited macro editor. Commands are inserted by choosing them from a master "command list," or by pressing the keys associated with each command. For example, to insert the command for applying bold to text, you press the F6 key.

In WordPerfect 6.1 for Windows, macros are stored in standard WordPerfect documents, and no special editor is needed to view and modify them. WordPerfect executes the commands contained in the document as a series of macro instructions.

Both approaches differ considerably from the technique used in Word 97. Word macros are written and developed using the integrated development environment of Visual Basic. This integrated programming environment runs in its own window, and it includes advanced debugging features, property-editing and code-editing features (including compile-time syntax checking and tools for constructing statements), an Object Browser, and code organization and tracking features.

Visual Basic is also shared by the other programs in the Microsoft Office 97, Professional and Developer Edition suites, including Microsoft Excel,

PowerPoint, and Microsoft Access. A single Visual Basic program can control any and all of these programs.

In this paper, the discussion is confined to converting WordPerfect macros to Word-only macros. However, you're not limited to using Word as the only element of automating your office solutions. You can write Word macros that control other Office 97 programs, as well as products from the over 40 companies that have currently licensed Visual Basic technology.

Deciding When a Macro Isn't Needed

Not all automated tasks require a macro in Word. Some of the macros created in WordPerfect may not need to be replicated in Word, because Word may offer a built-in feature that can do the same job. Instead of reconstructing macros, consider using the follow Word features:

- Templates – Standard document formats can be stored in template files. Whenever a new document is created based on a template, it inherits the formatting of the template. Templates are used to store Word macros projects, styles, AutoText entries, and command bar, menu and shortcut key customizations. To base a new document on a template, click **New** on the **File** menu.
- Forms – Word documents can include form elements such as text boxes, command buttons, and check boxes to create an online form. The forms feature of Word simplifies the task of filling out forms so a macro isn't needed. An added benefit of the form feature in Word is that the rest of the document (including text, images, and formatting) can be protected against editing. Form elements can be added using the **Control Toolbox** toolbar or the **Forms** toolbar.
- AutoText – The AutoText feature allows you to store and retrieve text, graphics, tables, and formatting. To use AutoText, click **AutoText** on the **Insert** menu.
- AutoCorrect – The AutoCorrect feature is used to automatically correct common misspellings, such as "teh" with "the." You can also use the feature to insert long-form text. As an example, type **sy**, and the AutoCorrect feature can "expand" it to *Sincerely Yours*. To use AutoCorrect, click **AutoCorrect** on the **Tools** menu.
- Insert Symbol – Use symbols with the powerful AutoCorrect feature in Word. AutoCorrect allows you to associate two or more characters with a given symbol. For example, typing the characters **(r)** inserts ® . To insert a symbol, click **Symbol** on the **Insert** menu.
- Styles – Repetitive formatting is efficiently handled by the style feature in Word and not a macro (a common technique with WordPerfect for DOS). With styles, you can easily redefine the formatting of a style and the text is automatically updated. To use styles, click **Style** on the **Format** menu.
- Letter Wizard – Business letter formatting can be applied using the Letter Wizard in Word. The Letter Wizard formats a letter based on your stylistic choices. If you need to create many letters using the same basic formatting, you can easily record a macro or create a letter template using the Letter Wizard. Whenever the macro runs, the formatting you chose is automatically applied. To use the Letter Wizard, click **Letter Wizard** on the **Tools** menu.
- Customized Menus, Toolbars and Shortcut Keys – Word lets you change toolbars and create shortcut keys for any command so that frequently-used features are just a mouse click or a keystroke away.

Understanding Where Macros Are Stored In Word

The standard WordPerfect storage method for macros is a file. Each macro is stored as a separate file on a disk, and is given a unique name. To run a macro, specify the name and WordPerfect reads the macro file.

In Word, macros are stored in documents and templates as Visual Basic *modules*. Macros are ordinarily stored in the user's default template, Normal.dot. However, Word allows you to store and use macros in any document or template. Additional templates can be loaded using the **Templates & Add-ins** dialog box (**Tools** menu).To run a macro from the **Macros** dialog box, you can choose to display all the available macros, or only those in a specific template or document.

Note WordPerfect 6.0 and later for Windows also supports macros in templates.

To share a macros with another user can pose a quandary if you're used to the WordPerfect approach to macros. Sharing Visual Basic macros is accomplished in a number of different ways.

- Give your Normal.dot file to the user.

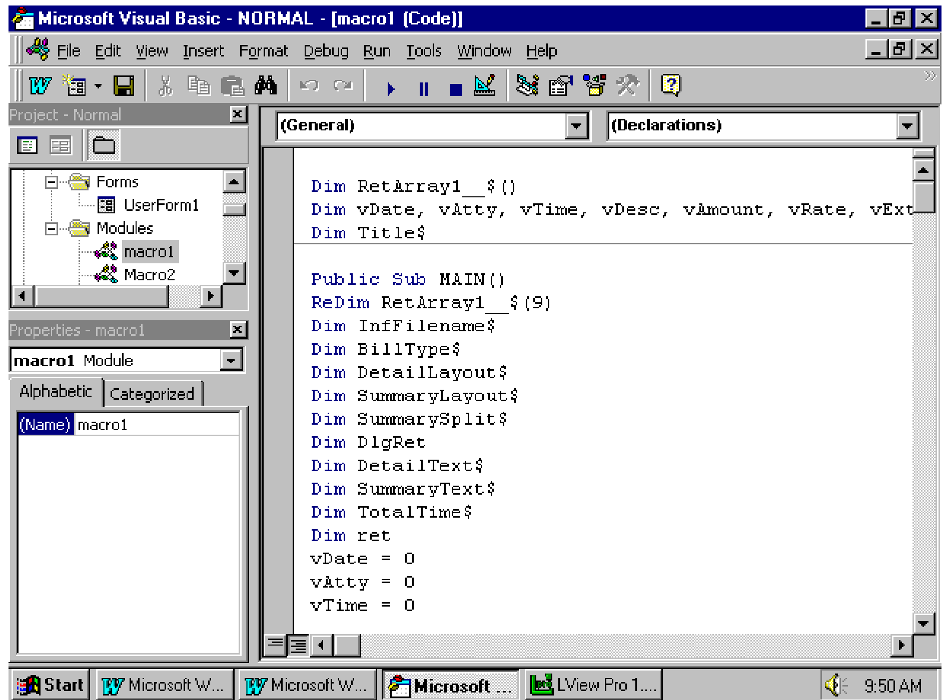
  Note This is not recommended because this will overwrite the user's existing Normal.dot file.

- Copy the Word macros to a separate template and give the template to the user (place the template in the user's \Program Files\Microsoft Office\Office\Startup folder; this makes all macros within the template automatically available whenever Word is used).
- Copy the macro to a document and distribute the document. The user can then use the Organizer to copy the macro to another template or document. The user can also copy and paste code between modules in the Visual Basic development environment.
- Export the Visual Basic module and distribute the file (*.bas). The file can be imported into another user's project using the **Import File** command (**File** menu).
- Place the template on a server and set the server path as the **Workgroup Templates** location (**File Locations** tab, **Options** dialog box, **Tools** menu).

The Visual Basic Editor: A Basic Roadmap

The heart of the macro feature in Word is the Visual Basic Editor. Before undertaking any conversion project, be sure to become well acquainted with the editor. The more you know about the editor, the faster and more successful you will be in converting WordPerfect macros.

Open the Visual Basic Editor by pointing to **Macro** on the **Tools** menu, and clicking **Visual Basic Editor** (or press ALT+F11). The editor, shown below, consists of a menu bar, a toolbar, and several windows.



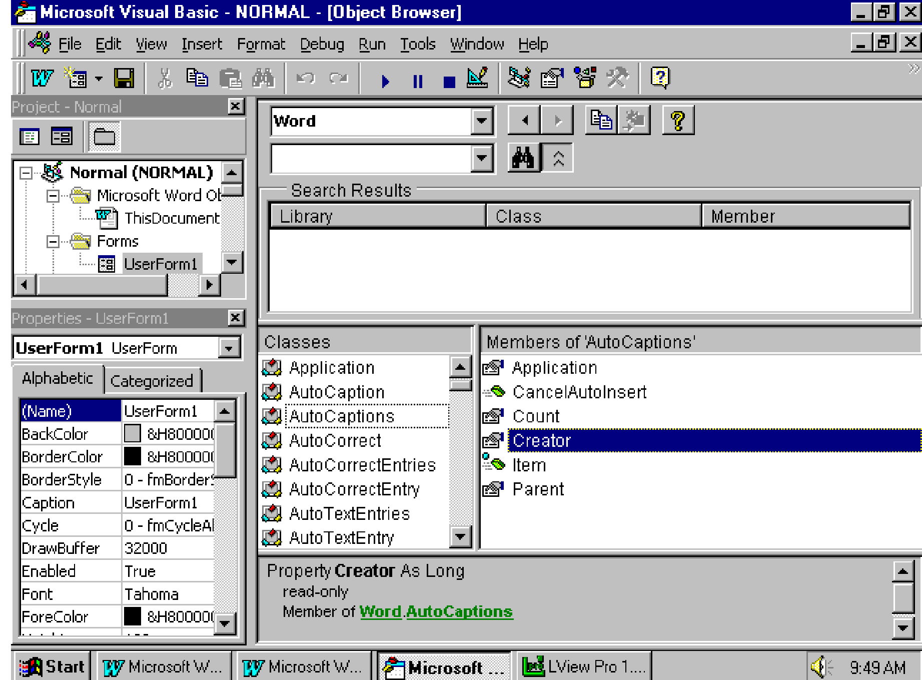
- The **Project Explorer** window displays all of the projects available for editing. (A project is analogous to macros stored in a WordPerfect template.)
- The **Properties window** displays all of the properties of the selected item. For example, when a command button on a form is selected, the **Properties** window shows options for the command button.
- The **Code window** displays the content of the current macro.

Other windows may appear depending on the options you've selected. For example, click **Object Browser** on the **View** menu to display a list of objects, properties, and methods you can use. Objects, properties, and methods can be combined to create Visual Basic instructions which are akin to programming commands in WordPerfect.

Understanding the Terminology of Visual Basic

Visual Basic uses a different set of programming terms than those used in WordPerfect. In Visual Basic, tasks are performed by applying properties and methods to objects. Objects are the fundamental building blocks of Visual Basic; almost everything you do in Visual Basic involves modifying objects. Every element of Word—documents, paragraphs, fields, bookmarks, and so on—is represented by an object in Visual Basic. To view a graphical representation of the object model for Word 97, see “Microsoft Word Objects” in Word Visual Basic Help. If you're not familiar with the terms object, properties, and methods, refer to the "Understanding objects, properties, and methods" topic in Word Visual Basic Help.

You can view the available objects, properties, and methods in the Visual Basic Object Browser. To see the Object Browser, display the Visual Basic Editor, and click **Object Browser** on the **View** menu. Select a library in the **Project/Library** box. The Word object library contains all the objects, properties, and methods (analogous to “commands” in WordPerfect) you need to write code using Visual Basic.



Comparing Syntax

Visual Basic and the WordPerfect macro language are very different. The differences of syntax between the two languages encompass the actual programming commands, and the way the commands are formatted to construct a working application.

Differences in syntax is the major stumbling block in converting WordPerfect macros to Visual Basic. The following table lists the major differences you will encounter when converting WordPerfect macros to Visual Basic.

Syntax Element	WordPerfect	Word
General		
Case sensitivity	WordPerfect commands and variables are not case sensitive. That is, the If command can be spelled if, If, or IF.	Statements, objects, varia Visual Basic are not case Basic Editor automatically conform to its standards.

Comments	WordPerfect for Windows:
	`// Commented line`
	WordPerfect for DOS:
	`{;}Commented line~`
		Apostrophe (') or REM cha
		line.

Common Programming Statements		

If statements	If always used with **EndIf**; **If** expression enclosed in parenthesis
	WordPerfect for Windows:
	`If (expression)`
	`    <statements if true>`
	`Else`
	`   <statements if false>`
	`EndIf`
	WordPerfect for DOS:
	`{IF}expression~`
	`   <statements if true>`
	`{ELSE}`
	`   <statements if false>`
	`{END IF}`
		If doesn't always need **En**
		enclosed in parenthesis; **I**
		`If expression Then`
		`   <statements if true`
		`Else`
		`   <statements if fals`
		`End If`
		– or–
		`If expression Then <st`
		– or–
		`If(expression, truepar`

For/Next loops	**For** or **ForNext** always used with **EndFor**
	WordPerfect for Windows:
	`ForNext (count; start; stop; step)`
	`   <repeating statements>`
	`EndFor`
	WordPerfect for DOS:
	{FOR}count~start~stop~step~~
	<repeating statements>
	{END FOR}
		For used with **Next** statem
		`For (counter = start t`
		`   <repeating statemen`
		`Next counter`

While loops	While used with EndWhile
	WordPerfect for Windows:
	`While (condition)`
	`   <repeating statements>`
	`EndWhile`
	WordPerfect for DOS:
	`{WHILE}condition~`
	`  <repeating statements>`
	`{END WHILE}`
		`While condition`
		`    <statements>`
		`Wend`

Repeat loops	Repeat used with Until.
	WordPerfect for Windows:
	`Repeat`
	`   <repeating statements>`
	`Until (condition)`
		Not directly supported. Do
		`Do Until condition`
		`  <repeating statement`
		`Loop`

Switch/Case	WordPerfect for Windows: Switch statement block used with **EndSwitch**; **CaseOf** statements define multiple tests for **Switch** expression.
	
	WordPerfect for Windows:
	Switch (expression)
	    CaseOf x: <statements>
	    CaseOf y: <statements>
	    CaseOf z: <statements>
	EndSwitch
	
	Note: WordPerfect for Windows also suppose a **Case** statement, but is rarely used.
	
	WordPerfect for DOS:
	{CASE}expression~
	   case1~label1~
	   case2~label2~
	   casen~label~~
		Select Case statement bl
		statements define multiple
		expression.
		
		Select Case expressior
		   Case x
		      <statements>
		    Case y
		       <statements>
		   Case z
		       <statements>
		End Select

Return statements	Returns from macro, subroutine, function, or procedure	Branches to and returns fr procedure. Not used to ret
Procedures and Functions		
Procedures	Block of self-contained code that does not return a value. Provided in WordPerfect for Windows only: Procedure Name (argument_list) <statements> EndProc	In Visual Basic, procedure by name. Procedures can subroutine procedure doe Sub Name (argument_lis <statements> End Sub

Functions	Block of self-contained code that returns a value. Provided in WordPerfect for Windows only:
	
	Function Name (argument_list)
	   <statements>
	   Return (expression)
	EndFunc
		Block of self-contained co
		
		Function Name (argumer
		   <statements>
		   Name=expression
		End Function

Calling routines	Routines are called using the **Call** statement (or in WordPerfect for Windows, implicitly by specifying the name of the routine.
	
	WordPerfect for Windows:
	Call (Routine_Name)
	   – or –
	Routine_Name(argument_list)
	
	WordPerfect For DOS:
	{CALL}Routine Name~
		Routines are called using
		implicitly by specifying the
		
		Call Routine_Name (arg
		   – or –
		Routine_Name (agument_

Control Flow		
Using line labels	Labels are referenced elsewhere in the macro by name. The macro can be directed to the label with a **Go** or **Call** statement. WordPerfect for Windows: Label (Label_name) WordPerfect for DOS: {LABEL}Label_name~	Labels are referenced els code by name or number. the label using a **GoTo** st GoTo Label_name Labels are created by typi beginning of the line, follo EndOfLoop:

Calling other macros	The **Chain**, **Run**, or **Nest** command executes other macros. With **Run/Nest**, execution returns to the calling macro when the called macro finishes.
	WordPerfect for Windows:
	`Nest (macro_name)`
	WordPerfect for DOS:
	`{NEST}macro_name~`
		Routines in other modules statement. Execution retur the called routine is finishe
		`Call Module_Name.Routi`

Error trapping	General error trapping with **Error** and **OnCancel** commands. No support for trapping specific kinds of errors.
	WordPerfect for Windows
	`OnError (Label)   // branch on error`
	`Error (Off!)  //turn off error trapping`
	WordPerfect for DOS
	`{ON ERROR}Label~  // branch on error`
	`{ON ERROR}~   //turn off error trapping`
		General error trapping with test for and trap specific e
		`On Error GoTo Label`
		`On Error GoTo 0   ' tu`
		Error routine code must be or function. You may also
		`On Error Resume N`
		`//continue to nex`
		`error`
		You can find out what erro **Err** object. See Visual Bas

Variables		
Variable scope	WordPerfect for Windows: By default, variables are visible only within the macro or procedure where they were defined. Global variables can be created using the **Global** command. WordPerfect for DOS: All variables are global, and remain in memory until specifically deleted, or when WordPerfect ends.	By default, variables are v procedure where they wer can be created using the **F** Variables can be visible w function, the current modu depending on declaration. See Visual Basic Help on possibilities.
Data typing	WordPerfect for Windows: all variables are variant type, able to contain any type of data. Supported data types include: 16-bit integers, 32-bit integers, 64-bit floating point, Boolean, and strings. WordPerfect for DOS: all variables are variant type. Supported data types include 16-bit integers and strings (under 128 characters) only.	Variables can be variant ty declared. Supported data integers, 64-bit float, Boole objects, and user-defined.

Persistent variables	WordPerfect for Windows:
	Persistent variables remain in memory after the macro ends. They are created with the **Persist** or **PersistAll** command.
	`Persist (VarName)`
	– or –
	`PersistAll`
	WordPerfect for DOS:
	All variables are persistent.
		Persistent variables can be Variables feature in Word. "Document Variables" in V stored in Document Variab document file and is availa open. Settings files are an data that must remain duri **PrivateProfileString** prop Basic Help). Global variab project is open.

Recording Macros to Learn Syntax

As mentioned earlier in this white paper, if you are fairly new to Visual Basic, a good way to learn Visual Basic syntax is to record a simple macro in Word. You can then examine the resulting Visual Basic code in the Visual Basic Editor.

1. In a blank Word document, start recording a macro by pointing to **Macro** on the **Tools** menu, and then clicking **Record New Macro**.
2. In the **Record New Macro** dialog box, type **testing** as the name of the macro, and click **Record**.
3. In the Word document type **This is a test**, press ENTER, and then click **OK**.
4. Stop recording by pointing to **Macro** on the **Tools** menu, and then clicking **Stop Recording**.

The macro is now recorded, and can be viewed in the Visual Basic Editor. To view the macro:

1. Point to **Macro** on the **Tools** menu and then click **Macros** (or press ALT+F8).
2. Select **testing** under **Macro name**.
3. Click **Edit**.

For more information on recording a macro, see “Recording a macro to generate code” in Word Visual Basic Help (use the **Find** tab to locate the topic).

The recorded macro appears in the code window as:

```
Sub testing()
'
' testing Macro
' Macro recorded 03/26/97 by Steve Masters
'
Selection.TypeText Text:="This is a test"
Selection.TypeParagraph
End Sub
```

The macro begins with a subroutine name, which by default is the same as the macro name. Comments (indicated by an apostrophe (')) provide a short description of the name of the macro, when it was recorded, and by whom. The actual recorded contents of the macro follow, and, in this case, consists of two commands:

```
Selection.TypeText Text:="This is a test"
Selection.TypeParagraph
```

The **Selection.TypeText** command specifies that text is to be inserted into the document. The **Selection.TypeParagraph** command specifies that a hard return is to be inserted in the document.

For comparison purposes, the same macro recorded in WordPerfect 5.1 for DOS appears in its macro editor as:

```
{DISPLAY OFF}
This is a test{Enter}
```

And in WordPerfect 6.1 for Windows:

```
Type (Text:"This is a test")
HardReturn()
```

As you can see, the syntax between these three word processors is vastly different, with thousands of possible permutations. Since it is impossible to enumerate all the differences between the macro languages, it’s generally easier to record new macros in Visual Basic, and use the new recordings as a basis for your new Word macros.

Converting Programming Commands

The following two tables summarize the programming keywords used with WordPerfect for DOS and WordPerfect for Windows, and compares them with their direct replacements in Visual Basic. Following these tables are several sections that detail specific conversion tasks, such as converting menus and dialog boxes, and converting macros that format and insert text.

WordPerfect for DOS to Visual Basic Command Cross-Reference

WordPerfect for DOS commands	Visual Basic commands

{;}	' or REM
{ASSIGN}	= assignment (ex: var = 1)
{BELL}	Beep
{BREAK}	None (Visual Basic Break command is used
{CALL}	Call
{CANCEL OFF} / {CANCEL ON}	On Error
{CASE} / {CASE CALL}	Select Case
{CHAIN}	None
{CHAR}	None (use MsgBox instead)
{DISPLAY OFF} / {DISPLAY ON}	Application.ScreenRefresh
{ELSE}	Else
{END FOR}	Next
{END IF}	End If
{END WHILE}	Wend
{FOR}	For
{GO}	GoTo
{IF}	If
{IF EXISTS}	IsNull
{KTON}	Asc (Or AscW for Unicode characters)
{LABEL}	LabelName:
{LEN}	Len
{LOOK}	KeyDown or KeyPress
{MID}	Mid
{NEST}	None (referencing the macro by name runs i
{NEXT}	Next
{NTOK}	Chr (or ChrW for Unicode characters)
{ON CANCEL}	On Error
{ON ERROR}	On Error
{PAUSE}	None
{QUIT}	Stop
{RETURN}	Return
{RETURN ERROR}	On Error
{STATE}	Various properties of Word objects
{STEP ON}/ {STEP OFF}	Break
{SYSTEM}	Various properties of Word objects
{TEXT}	InputBox
{WAIT}	Declare Sub Win_Sleep Lib "Kernel32" Alias (ByVal dwMilliseconds As Long (then use Sleep in the code when WAIT was
{WHILE}	While
{Enter} (to insert new line)	Selection.TypeParagraph

{Tab} (to insert a tab)	Selection.TypeText Text:=vbTab
“Text here” (to insert text)	Selection.TypeText Text:="Text"

WordPerfect for Windows to Visual Basic Command Cross-Reference

WordPerfect for Windows commands	Visual Basic commands

// Comment	' or REM
AppActivate	AppActivate
AppExecute	Shell
AppLocate	None (not needed)
Assign	= assignment (ex: var = 1)
Beep	Beep
Break	None (Visual Basic Break command is used
Call	Call
Cancel	On Error
Case / Case Call	Select Case
Chain	None
CTON	Asc (or AscW for Unicode characters)
DDE commands	None (superceeded by Visual Basic and OLE
Dimensions	UBound (also LBound to get lower bounds)
Discard	Erase (used for for arrays)
DLL commands	Declare Sub or Declare Function
Else	Else
EndFor	Next
EndFunc	End Function
EndIf	End If
EndProc	End Sub
EndSwitch	End Select
EndWhile	Wend
Error	On Error
Exists	IsNull
For	For
ForEach	For Each
ForNext	For
Function	Function
GetNumber	InputBox
GetString	InputBox
GetUnits	InputBox
Global	Public
Go	GoTo
If	If
Include	None
Indirect	None
Label	LabelName:
Local	Private
MenuList	None

MessageBox	MsgBox
Nest	None (referencing the macro by name makes
Next	Next
NotFound	On Error
Ntoc	Chr (or ChrW for Unicode characters)
NumStr	Str
OnCancel	On Error
OnError	On Error
OnNotFound	On Error
Pause	None
Persist	None
Procedure	Sub
Prompt	None
Quit	Stop
Repeat	None (use Do/Loop)
Return	Return
Run	None (referencing the macro by name makes
SendKeys	SendKeys
Speed	None (use step through with debugger)
StrLen	Len
StrNum	Val
StrPos	InStr
StrUnit	None
SubStr	Mid
Switch	Select Case
ToLower	LCase
ToUpper	UCase
Until	None (use Do/Loop)
Use	None
VarErrChk	IsNull
Wait	None
While	While
HardReturn() (to insert new line)	Selection.TypeParagraph
Tab() (to insert a tab)	Selection.TypeText Text:=vbTab
Type ("Text here") (to insert text)	Selection.TypeText Text:="Text"

Converting Variable Assignments

There is a great deal of similarity in assigning variables with WordPerfect macros and Visual Basic. Both use this common construction:

```
VarName = value
```

where value can be a number, a string, another variable, or a unique variable type, such as a Boolean True/False.

Where Visual Basic differs from WordPerfect is in global declarations of variables, and in the scope of variables.

- In WordPerfect for DOS, variable assignments use the **{ASSIGN}** command:

  ```
  {ASSIGN}var~value~
  ```

  These must be converted to the VarName = value syntax.
- In WordPerfect for Windows, the **Assign** command can be used as an optional method for variable assignment. As with WordPerfect for DOS, these assignments must be rewritten using the VarName = value syntax.
- In WordPerfect for DOS, variables remain in memory until they are specifically deleted, or when WordPerfect ends. In Visual Basic, variables are retained only as long as the macro runs.
- In WordPerfect for Windows, you define a Global variable (a variable that can be shared between macros and procedures) using the **Global** command. In Visual Basic, you use the **Public** keyword to define a global variable. In Visual Basic, variables are global only within the macro that created them.

For more information on defining global variables in Visual Basic, see the **Public** keyword topic in Visual Basic Help.

Converting Expressions

Expressions are used with programming statements such as **If** and **While** to evaluate some condition. Expression syntax is almost identical between WordPerfect for Windows and Visual Basic, except that in Visual Basic, parentheses are not required for use with most statements that use expressions.

For example, in WordPerfect a typical **If** expression may appear as:

```
If (MyVar = "1")
```

In Visual Basic, the same expression appears as:

```
If MyVar = 1 Then
```

Note that the parentheses are dropped, and that the **If** statement is used with the **Then** keyword.

Expressions appear differently in WordPerfect for DOS. The typical **If** statement may look like this:

```
{IF}"{VARIABLE}MyVar~"="1"~
```

See above for an example of how this expression must be rewritten for Visual Basic. In particular, there is no need to use the **{VARIABLE}** command (Visual Basic understands **MyVar** is a variable), nor is there a need to surround the variable and tested value with quotes.

Converting Macros that Insert and Format Text

Inserting and formatting text is a common task for many WordPerfect macros. In WordPerfect for DOS, inserting text in a document requires only that the text be entered in the macro. No special commands are needed to insert text. For example, the following WordPerfect for DOS macro inserts the name “Benjamin Franklin” at the current insertion point:

```
{DISPLAY OFF}
Benjamin Franklin
```

In WordPerfect for Windows, text is inserted using the **Type** command.

```
Type ("Benjamin Franklin")
```

In Visual Basic, the statement to insert text at the current insertion point is:

```
Selection.TypeText Text:="Benjamin Franklin"
```

- **Selection** is the Word object that refers to the selection in a document window pane.
- **TypeText** is the method of the **Selection** object for inserting text
- **Text:=** is the required parameter for the **TypeText** method.
- **Benjamin Franklin** is the text to insert.

The **Selection** object can be used to insert text, move the selection, and edit text around the selection (the selection can refer to the insertion point or the selected text).

Command	WordPerfect for DOS	WordPerfect for Windows	Word/Visual I
Text	Text	Type ("Text")	Selection.Type
Tab	{Tab}	Tab()	Selection.Type
Indent	{Indent}	Indent()	Selection.Para = InchesToPoi
New line	{Enter}	HardReturn()	Selection.Type
Page break	None.	HardPageBreak()	Selection.Inser
Bold on/off	{Bold}	AttributeAppearanceToggle (Bold!)	Selection.Font.
Underline on/off	{Underline}	AttributeAppearanceToggle (Underline!)	Selection.Font.
Go to start of document	{Home}{Home}{Up}	PosDocTop()	Selection.Hom
Go to end of document	{Home}{Home}{Down}	PosDocBottom()	Selection.EndK
Go to start of line	{Home}{Left}	PosLineBegin()	Selection.Hom
Go to end of line	{End}	PosLineEnd()	Selection.EndK
Delete next character	{Delete}	DeleteCharNext()	Selection.Delet Count:=1
Backspace	{Backspace}	DeleteCharPrev()	Selection.Type
Delete next word	{Block Move}	DeleteWord()	Selection.Delet
Delete previous word	{Del Word}	DeleteWord()	Selection.Delet
Insert date as text	{Date/Outline}1{Enter}	DateText()	Selection.Inser DateTimeForm InsertAsField:=
Insert date as updatable code	{Date/Outline}2{Enter}	DateCode()	Selection.Inser DateTimeForm InsertAsField:=

Converting Macros that Use Documents

Many macro tasks entail opening, closing, and saving documents. These tasks are readily duplicated in Visual Basic.

Opening a file is one of the most common tasks performed by a macro. In WordPerfect for DOS, documents are opened with a macro by duplicating the

keystrokes for retrieve: press SHIFT+F10, type the file name, and press ENTER. In a macro, the commands may appear as:

```
{DISPLAY OFF}
{Retrieve}myfile.txt{Enter}
```

In WordPerfect for Windows, the **FileOpen** command is used to open a file. In a macro the command appears as:

```
FileOpen ("filename.wpd")
```

where *filename.wpd* is the name (and optionally, the path) of the document to open.

In the Word implementation of Visual Basic, the **Open** method is used to open an existing document. This method has many variations and uses, but the most common is

```
Documents.Open FileName:="filename.doc"
```

where *filename.doc* is the name (and optionally, the path) of the document to open.

Macros are also typically used to save a document once it has been edited, either by macro or by a user. In WordPerfect for DOS, a macro that saves a file contains the commands:

```
{Save}filename.ext{Enter}
```

(Note: WordPerfect warns you if the file already exists. If it does, the macro needs to supply the **Y** keystroke to answer Yes.)

In WordPerfect for Windows, the **FileSave** command is used for the same purpose:

```
FileSave ()   // Save an already named file
```

– or–

```
FileSave ("filename.wpd")   // Name and save a file
```

The Word implementation of Visual Basic uses the **SaveAs** or **Save** method for saving a file, depending on whether the file has already been previously saved, and so already has a name. For example, to save the current document, giving it the name "Mydoc.doc," use:

```
ActiveDocument.SaveAs Filename:="Mydoc.doc"
```

If the document already exists, you can the **Save** method instead. Assuming Mydoc.doc has already been saved at least once:

```
Documents("Mydoc.doc").Save
```

Both the DOS and Windows versions of WordPerfect automatically open a blank window for creating new documents. Word does not open a blank window in preparation for a new document; you must explicitly tell it to do so. Therefore, if you wish to create a document in a new blank window, you should always precede any document creation activity with an **Add** method. The following creates a new document, which can then be edited and saved, the same as any other Word document:

```
Documents.Add
```

Finally, in WordPerfect for DOS, a document is closed in a macro by replicating the Exit document keystrokes, which is F7, **n**, **n**. (The two n's answer No; you don't wish to save the document -- assuming you've previously saved it -- and you don't wish to exit the WordPerfect program.) In a macro, the keystrokes appear as:

```
{Exit}nn
```

In WordPerfect for Windows, the **Close** command is used in a macro to close a document. As with the WordPerfect for DOS example above, it is assumed the document has already been saved. In a macro the command appears as:

```
Close ()
```

The Close method is used in the Word implementation of Visual Basic to close a document. To close a document with Visual Basic, you provide a statement such as:

```
Documents("myfile.doc").Close
```

where *myfile.doc* is the file name. If the document has not yet been saved, Word reminds the user to save the file. To avoid this prompt, you can first save the document using the **Save** method, detailed above, or by adding the **SaveChanges** argument to the **Close** method, as shown here.

```
Documents("myfile.doc").Close SaveChanges:=wdSaveChanges
```

Converting User Input

A common requirement of most macros is to display a message to the user and wait for a response. Sometimes the response is a single mouse-click or keystroke denoting “OK”; other times it’s a Yes/No response, or perhaps a full text answer.

WordPerfect for Windows offers a number of built-in commands for displaying a message and waiting for user feedback. These are:

- **MessageBox** – Displays a message, with an optional assortment of buttons (**OK**, **OK/Cancel**, **Yes/No**, and so forth). The **MessageBox** command returns the value of the button pressed as a numeric value.
- **GetString** – Displays a message and waits for the user to type a response.
- **Prompt** – Displays a message box that does not wait for a user’s response (either **OK** or **Cancel**). Most often used in conjunction with the **Pause** command, which temporarily pauses the macro.

WordPerfect for DOS offers more rudimentary user interaction -- three commands that normally display text only in the status line at the bottom of the screen.

- **{CHAR}** – Display a text message and wait for the user to press a key (most often used for Yes/No responses).
- **{TEXT}** – Display a text message and wait for the user to type a response.
- **{PROMPT}** – Display a message that does not wait for a user response. Most often used in conjunction with the Pause command.

Visual Basic offers similar functionality as the above WordPerfect commands, but as statements.

- **MsgBox** – Functionally equivalent to MessageBox in WordPerfect for Windows.
- **InputBox** – Functionally equivalent to GetString in WordPerfect for Windows.

Converting Macros that Pause

Unlike WordPerfect, Visual Basic lacks a means to pause execution in the middle of a process. This is not a serious shortcoming; it merely requires that the macro behave in a different manner than it did in WordPerfect.

As an example, a common WordPerfect macro that used pauses is the "memo fill-in helper." This macro paused for user entry for each line of the memo. There is no need to replicate this behavior in Word, and in fact, it is undesirable to do so. It is far better to write a macro that collects all of the pieces of information for the memo, then inserts all the text at one time.

For example, here is a short macro that asks for the To:, From:, and Subject: fields of a memo, then inserts the information after all three prompts have been answered.

```
ToField = InputBox("Send memo to")
FromField = InputBox("Memo from")
SubjectField = InputBox("Memo about")
With Selection
    .TypeText Text:="To: " & ToField
    .TypeParagraph
    .TypeText Text:="From: " & FromField
    .TypeParagraph
    .TypeText Text:="Subject: " & SubjectField
    .TypeParagraph
End With
```

Converting Alerts

Alerts are commonly used to communicate important information to the user, such as an error or a reminder. In WordPerfect for DOS, alerts were typically created using the {CHAR} command, which allows the macro to pause temporarily and display a message in the status prompt (or elsewhere on the screen, using additional screen-placement characters). Pressing a single key releases the pause, and the macro continues.

In WordPerfect for Windows, both the **MessageBox** command the **Prompt** command typically used to display an alert to the user. The message box temporarily pauses the macro; pressing **OK** or the ENTER key closes the box, and restarts the macro. A **MessageBox** alert may look like the following:

```
MessageBox (; "Title"; "This is an alert!")
```

Similarly, the **MsgBox** statement in Visual Basic can be used to display an important message to the user. As with the **MessageBox** command in WordPerfect for Windows, **MsgBox** temporarily pauses the macro. Pressing **OK** or the ENTER key closes the box, and restarts the macro.

```
MsgBox "This is an alert!"
```

Converting Dialog Boxes and Menu Lists

Visual Basic fully supports Windows, including the ability to create custom dialog boxes. Dialog boxes are most often used to collect information from the user; for example, their name, address, and phone number. Click the **OK** button, and the Visual Basic code reads the values provided in the dialog box, and uses them accordingly. The code may use the name, address, and phone number to create a custom letter, or to fill out a simple database stored in Word document format.

WordPerfect for DOS lacks a means to create dialog boxes. Rather, complex menus and other elements of user interface must be created “from scratch” in

WordPerfect for DOS. One common method to create a screen menu in WordPerfect for DOS is with the **{PROMPT}** command, positioning characters (including line-draw characters) to produce the image of a pop-up box or menu. Any WordPerfect macro programmer who has taken the time to construct such a menu knows the amount of time and effort required.

The WordPerfect for Windows macro system supports dialog boxes. A simple dialog box editor comes with the program (version 6.1 and later). With this editor, you can construct a dialog box by dragging controls (text boxes, push buttons, and so forth) onto a blank dialog box template. The definition for the dialog box is contained in a separate portion of the macro file, and is not readable, except by WordPerfect.

Visual Basic provides a fully functional and sophisticated dialog box editor. This editor allows you to build almost any dialog box (known as a UserForm in Visual Basic Editor) by selecting controls from a palette, and placing them on the dialog box.

Each UserForm control supports a series of "events," such as clicking or double-clicking (the events are different for each type of control). Using event routines, you can perform a specific action when the user chooses a control on the user form. To create a UserForm, click **UserForm** on the **Insert** menu in the Visual Basic Editor. This action creates a blank UserForm and displays a ToolBox window with a number of UserForm controls.

These controls include:

- **ComboBox** – In its typical form, combination text box and list box.
- **CheckBox** – Non-exclusive option selection; click to turn the option on or off.
- **CommandButton** – A push button used to initiate an event.
- **Frame** – Creates a functional and visual grouping of controls.
- **Label** – Static text that doesn't change. Use labels for explanatory text.
- **OptionButton** – Shows the selection status of an item. Note that each **OptionButton** in a **Frame** control is mutually exclusive.
- **SpinButton** – Entry box for specifying values; up/down push buttons lets you select a value with the mouse.
- **TabStrip** – Collection of tabs for selecting different sets of options in a dialog box.
- **Image** – Displays a picture.
- **TextBox** – Entry blank for writing text. You can make the box almost any size you want.

For more information on a particular type of control, add one to a form, select it, and press F1.

For information on adding controls and otherwise customizing the **ToolBox**, press F1 with a form selected, click **Help Topics**, switch to the **Contents** tab, and expand **Microsoft Forms Design Reference**.

Once you have added a control to a UserForm, you can write code that will run when one of its events is triggered. For example, you can write an event procedure that will run every time a button is clicked, as shown below.

```
Private Sub CommandButton1_Click()
    MsgBox "The button was clicked"
End Sub
```

To write an event procedure for a control, double-click the control to display the code associated with the contrel in the **Code** window. In the **Procedure** drop-

down list, click the event you want to write a procedure for. The code you write in this procedure will automatically run when the specified event occurs on the specified control.

For more information on creating dialog boxes, see the UserForm topics in Visual Basic Help or refer to Chapter 12, "ActiveX Controls and Dialog Boxes", of the Microsoft Office 97/Visual Basic Programmer's Guide.

Converting Yes/No Messages

A common requirement in a WordPerfect macro is to display a message and ask for a Yes/No response. In WordPerfect for DOS, this is often accomplished with the **{CHAR}** command, followed by an **{IF}** test. Example:

```
{CHAR}key~Do you want to continue (Y/N)?~
{IF}"{VARIABLE}key~"="y"~
                                      {;} Yes~
{ELSE}
                                      {;} No~
{END IF}
```

In WordPerfect for Windows, asking for a Yes/No response is often accomplished using the **MessageBox** command. This command displays a message box with **Yes** and **No** buttons. An **If** test determines which button was clicked (the value “6” means the **Yes** button was clicked):

```
MessageBox (Ret; "Continue"; "Do you want to continue?")
If (Ret=6)
                                      // Yes
Else
                                      // No
EndIf
```

Use the **MsgBox** function to ask Yes/No questions in Visual Basic. Follow with an **If** test to determine which button – **Yes** or **No** – was clicked by the user. Remember that the **MsgBox** function can display other button sets, depending on the options used. The following example shows how to use the **MsgBox** function to display a message box with **Yes** and **No** buttons. Alternative options display the **OK** button only, **Yes/No/Cancel**, and other button variations.

```
Ret = MsgBox (Prompt:="Do you want to continue?", Buttons:=vbYesNo)
If Ret = vbYes Then
                                      ' Yes
Else
                                      ' No
End If
```

Converting DLL Calls

The macro language in WordPerfect for Windows provides access to Windows API (application programming interface) routines. These routines are typically contained in one or more dynamic-link libraries (DLLs). Windows itself is composed of numerous DLLs; most Windows applications have or support additional DLLs of their own.

In WordPerfect for Windows, many functions contained in DLLs can be used by a macro. The DLL was first “registered” by WordPerfect, and the desired function within the DLL was called. A typical DLL call may look like the following:

```
DLLLoad (User; "USER") // Load Windows USER.EXE DLL
DLLCall (User; "MessageBeep"; ret:WORD; {})  // Call MessageBeep function
DLLFree (User)  // Release DLL registration
```

This example calls the **Message** function, which is contained in the User.exe file, one of three primary DLLs used in Windows 3.1. (The purpose of the

MessageBeep function is irrelevant for this discussion, but for the curious, the function sounds the computer's warning chime.)

In Visual Basic, DLLs are registered using a **Declare Sub** or **Declare Function** statement.

- Use **Declare Sub** when the DLL function does not return a value, or you don't care what value is returned.
- Use **Declare Function** when the DLL function returns a value that you want to get.

To use **Declare Sub** or **Declare Function**, you must also provide the name of the routine you want, the name of the DLL that contains the routine, and the list of arguments, if any, for the routine. Using **MessageBeep** as an example, the API reference for Visual Basic appears as:

```
Declare Sub MessageBeep Lib "User" (ByVal N As Integer)
```

Declare Sub and **Declare Function** statements are placed in the Declaration portion of the code for the module. This locates the statement before any other Subs or Functions in the module, and makes it globally available within any Sub or Function.

Note The User DLL referenced above is a 16-bit file. Word 97 and all other programs in the Microsoft Office 97 suite are 32-bit applications, and therefore require the use of 32-bit routines. For the Visual Basic, use the User32 DLL file instead.

Once the API function has been declared, it can be called from anywhere in the module. The function is called by name; any parameters needed by the API function are provided. For example, to call the **MessageBeep** function, you provide the following somewhere in module code:

```
MessageBeep (1)
```

For more information on Windows API and using DLLs, see the **Declare** statement in the Visual Basic Help, and consult books on Windows programming. *Visual Basic Programmers Guide to the Windows API*, published by Microsoft Press, is an excellent resource for learning more about the Windows API. Additional information on porting 16-bit DLL/API calls to corresponding 32-bit calls can be found on the Office Developer Forum (http://www.microsoft.com/officedev/TechInfo/techinfo.htm)

Converting Arrays

WordPerfect for Windows supports variable arrays (arrays are not supported in WordPerfect for DOS). Arrays can be created in WordPerfect for Windows two ways:

Method 1:

```
Declare (ArrayName[size]) // size is number of elements in array
ArrayName[1]="value 1"
ArrayName[2]="value 2"
…
ArrayName[n]="value n"
Method 2:
ArrayName[]={"value 1"; "value 2"; … ; "value n"}
```

Arrays are created by first declaring them, and then filling each element with values, similar to Method 1, above. Instead of using brackets around the index values for the array, Visual Basic uses parentheses. And, by default, arrays in Visual Basic are zero-based; they are 1-based in WordPerfect for Windows.

That is, in WordPerfect for Windows the first array element starts at 1. In Visual Basic, the first array index is 0 (unless you use the Option Base 1 statement to specify that the first array index is 1). Here is an example:

```
Dim MyArray(10) 'creates array with 11 elements (0-10)
MyArray(0) = "value 1"
MyArray(1) = "value 2"
```

… and so forth.

Unlike WordPerfect for Windows, Visual Basic supports dynamic resizing of arrays after they have been defined. This allows you to make an array larger (add more elements) while the code is running. You may resize the array with the **ReDim** statement, or by defining a *dynamic array*.

In WordPerfect for Windows, each element of an array can store different kinds of data. You can mix elements with integers with elements with strings, for example. Data typing is stricter in Visual Basic, as this lowers the memory overhead required to store the array. To specify an array that can contain any mix of data types, declare the array without a data type definition, as in:

```
Dim MyAnythingArray(10)
```

This defines an array with *variant* elements; that is, each element can accept any data type.

However, if you know that an array contains only a certain type of data, it is usually better to define its data type. This saves memory and makes the array more efficient. The following creates an array that can contain only integers in each element.

```
Dim MyIntegerArray(10) As Integer
```

Improving upon WordPerfect Macros

There is no reason to precisely duplicate a WordPerfect macro for use with Visual Basic. You can enhance the features, functionality, and performance of the original macro by taking advantage of the extra features available in Visual Basic. These include enhanced string functions, enhanced math functions, file and directory functions, registry statements, and properties for obtaining current values from Word.

Additional String Functions in Visual Basic

Visual Basic supports a rich variety of string functions that in WordPerfect requires sophisticated macro coding. These functions including trimming extra spaces from the beginning or ending of a string, returning just a certain number of characters from the beginning or ending, and many more.

Of the following additional string functions, **Format**, **Like**, and **StrComp** offer extraordinary functionality, and can often be used to reduce pages of WordPerfect macro code to a simple one-line statement.

String Function	What it does

Format	Format a string with user-defined formatting rules
Left	Returns specified number of characters from beginning of string
Like	Compare a string to a pattern.
LSet	Justifies a string with left alignment.
LTrim	Trims spaces off beginning of string
Right	Returns specified number of characters from end of string
RSet	Justifies a string with right alignment
RTrim	Trims spaces off end of string
Space	Create a string variable with *x* spaces
StrComp	Compare two strings (greater than, less than, equal to)
StrConv	Converts strings to other formats (including other character sets)
String	Create a string variable with *x* characters
Trim	Trims spaces off beginning and end of string

Additional Math Functions in Visual Basic

These built-in functions allow you to perform complex math equations, not possible (or readily possible) with the standard +. -. *, and / arithmetic operators.

Math Function	What it Does
Abs	Returns absolute value of a number
Atn	Returns the arctangent of an angle
Cos	Returns the cosine of an angle
Exp	Returns value of e (the base of natural logarithms) raised to a power
Fix	Returns integer portion of a number
Int	Returns integer portion of a number
Log	Returns the natural logarithm of a number
Rnd	Returns a random number
Sng	Returns the sign (+ or -) of a number
Sin	Returns the sine of an angle
Sqr	Returns the square root of a number
Tan	Returns the tangent of an angle

Registry Statements

Windows 95, and Windows NT 4.0 use the Windows registry to store information about applications. This registry is accessible through Visual Basic. You can read and set values for other programs, as well as read and set values for your own Visual Basic code.

Registry Function	What it Does
GetSetting	Returns a key setting value from an application's entry
GetAllSettings	Returns a list of key settings and their respective values
DeleteSetting	Deletes a section or key setting from an application's entry
SaveSetting	Saves or creates an application entry

File Functions

Visual Basic supports a number of highly useful functions for working with files. You can obtain file names, file size, file attributes, file creation dates and times, and other information directly in Visual Basic.

File Function	What it Does
Dir	Returns name of a file, directory, or folder that matches a specified patte attribute, or the volume label of a drive
FileAttr	Returns value representing the file mode for files opened using the Open statement
FileCopy	Copies a file
FileDateTime	Returns the date and time when a file was created or last modified
FileLen	Returns value specifying the length of a file in bytes
GetAttr	Returns a value representing the attributes of a file, directory, or folder
Input	Returns a string of characters from a file
Kill	Deletes files from a disk
Open	Enables input/output (I/O) to a file
Print	Writes display-formatted data to a sequential file
SetAttr	Sets attribute information for a file

Obtaining and Setting Current Values from Word

One of the limitations of WordPerfect macros is that the language lacks extensive commands for retrieving the current value from WordPerfect itself. For example, there is no direct way in WordPerfect to determine the value of most user settings that may affect the way a macro operates.

With Visual Basic, it's possible to query the state of Word to determine a current setting. The **AllowFastSave** property of the **Options** object, for example, contains the current setting for the **Allow Fast Saves** setting in the **Options** dialog box. To obtain this value in Visual Basic, use the **Options** property (which returns the **Options** object), a period, and the **AllowFastSave** property:

```
Ret = Options.AllowFastSave
```

The value (true or false) is stored in the Ret variable.

Similarly, the Word environment can be changed by setting a property value, assuming the property is not read-only. Therefore, to change the **Allow Fast Save** option, specify True or False, depending on whether you want the option turned on or off. In the following example, the **Allow Fast Save** option is turned off.

```
Options.AllowFastSave = False
```

Communication with Other Applications

Visual Basic Applications supports Automation (formerly known as OLE Automation). Automation is a feature of the Component Object Model (COM), an industry-standard technology that applications use to allow other applications to control them. For example, a word processor may control a spreadsheet program in order to pull out values from a worksheet, chart, cell, or range of cells, and place those values in a document.

When an application supports Automation, the objects the application exposes can be accessed by Visual Basic. Visual Basic manipulates these objects using methods of the objects supported by the program, or by getting and setting the properties of an object. For example, you can create a Word Automation object named MyObj and access the properties and methods of the Automation object (the Application object in Word) as shown below.

```
With MyObj
    .Selection.InsertAfter Text:="Hello, world."    ' Insert text
    .Selection.Font.Bold = True                     ' Format text
    .ActiveDocument.SaveAs "C:\WORDPROC\DOCS\TESTOBJ.DOC" 'Save document
End With
```

Use the following Visual Basic functions to access an Automation object:

Function	Description
CreateObject	Creates and returns a reference to an Automation object.
GetObject	Returns a reference to an existing Automation object.

Each program that supports Automation provides documentation on the objects, properties, and methods that can be accessed. The objects, functions, properties, and methods supported by an application are usually defined in the application's object library.

Note that programming inside of Word is the same as programming outside of Word (from Microsoft Excel or Visio for example). In both cases you use Visual Basic and work with the Word object model. For more information on automating Word objects from other applications, see the "Communicating with other applications" topic in Word Visual Basic Help or Chapter 7, "Microsoft Word Objects", of the Microsoft Office 97/Visual Basic Programmer's Guide.