The Microsoft Excel 5.0 Development Platform

I. Introduction

Microsoft Excel version 5.0 introduces a new paradigm in applications software development by providing the first object oriented programming environment available in a spreadsheet. In version 5.0, Microsoft has, essentially, objectified Excel by breaking it up into 128 separate programmable objects. At the same time, Microsoft has introduced a new macro language in Excel, Visual Basic for Applications, a powerful programming language that provides the tools for integrating Excel objects into custom solutions. Excel 5.0 also provides support for Object Linking and Embedding (OLE) version 2.0, making it easy to tie together components of other Microsoft Office applications, as well as third-party applications that support OLE 2.0. Additionally, Excel 5.0 provides full support for Windows Open Services Architecture (WOSA) data transfer protocols, including Open Database Connectivity (ODBC), Microsoft's Messaging Applications Programming Interface (MAPI) and Microsoft's Telephony Applications Programming Interface (TAPI). With it's advanced set of powerful programmable objects, Visual Basic for Applications and support for OLE 2.0, Microsoft Excel version 5.0 offers one of the richest development environments for the Windows, Windows NT and Macintosh platforms.

There are currently several major Fortune 500 companies that have custom solutions developed in Excel in widespread use. Additionally, there are numerous Excel applications that are commercially available for a variety of uses. Excel has been recognized throughout several industries as a flexible, powerful and easy-to-use data analysis tool, and companies across the globe are tapping into Excel's power by not only adopting Excel as a spreadsheet standard, but also by taking advantage of Excel's development capabilities to create custom applications. Excel, being a spreadsheet, is a very general tool, and as such, has been adopted for a wide variety of specific uses in numerous industries - financial, general business, engineering, medical, educational, real estate, insurance, accounting, etc. Companies are starting to realize that efficiencies are to be gained by customizing the spreadsheet environment for specific tasks - and through such realization, have started to take advantage of the customization and development power within Excel. Now with Excel version 5.0, Microsoft has taken a major step forward in offering the world's most advanced development environment available in an applications program without sacrificing ease-of-use. This document is intended to provide an introduction to Excel 5.0 from a custom application development perspective, providing both a high-level summary of the Excel development environment, as well as a technical overview of the Excel Object Model and Visual Basic for Applications.

II. Why develop with a spreadsheet?

While using a spreadsheet program to develop custom applications may be a new notion to many developers, it has several advantages to offer:

- **Excel 5.0 Offers a Rich Set of Powerful, Programmable Objects.**
- **Excel 5.0 Offers an Advanced Programming Language - Visual Basic for Applications.**
- **Excel 5.0 Provides Support for Object Linking and Embedding (OLE) ver. 2.0.**
- **Excel 5.0 Offers Support for WOSA Data Transfer Protocols - ODBC, MAPI and TAPI.**
- **Excel 5.0 is the Market-Leading Spreadsheet Across All Platforms**

Let's take a look at each of these advantages in greater detail.

1. Excel 5.0 Offers a Rich Set of Powerful, Programmable Objects

By breaking Excel 5.0 up into programmable objects, Microsoft is giving developers access to the Excel 5 code. Microsoft has invested hundreds of person-years in developing the Excel 5 code base and has utilized the efforts of some of the world's best software developers in writing the advanced algorithms that make up Excel. With Excel 5.0, it is now possible for third-party developers to leverage this investment made by Microsoft by integrating Excel objects into their own custom applications.

Developers Can Incorporate Excel Objects into their Code

In the past, if a developer required, say, a charting module in a custom application, the developer would have to write the charting routines him/herself - such a task is quite daunting considering the numerical computation involved and the advanced algorithms required for graphics routines in creating a chart. With Excel 5.0, a developer can bypass this burden and tap directly into the Excel Chart object - by merely passing data to the Chart object by using a few simple commands, Excel will do all the work in computing numbers and handling graphics routines, saving the developer a tremendous amount of time and effort in creating more powerful applications.

Excel offers several other very powerful objects which are easy to tap into. Perhaps, Excel's most powerful object is the PivotTable object, which provides a mechanism for taking large amounts of database data and placing it in a logical, multi-dimensional spreadsheet table. The technology behind the PivotTable object allows for lightening fast queries and provides an easy-to-use interface that gives users the ability to actually re-orient, or pivot, data by dragging database field labels with the mouse. Other objects within Excel include workbooks, worksheets, menus, toolbars, dialogsheets, on-sheet controls (dropdown listboxes, option boxes, check boxes), etc. - all of the tools necessary for creating advanced applications.

The objects in Excel 5.0 are ideally suited for creating a wide variety of different types of custom applications. Several large corporations have adopted Excel solutions for Enterprise Information Systems - systems that provide an intuitive interface for interpreting large amounts of corporate data. Because of its support for ODBC, Excel can read data from a number of different database formats. Once data is in Excel, there are numerous powerful tools for data analysis and manipulation that Excel has to offer - tools that include PivotTables, over 400 advanced

worksheet functions, sorting routines, subtotaling algorithms, charting, etc. Let's take a look at a diagrammatic view of what a typical Excel EIS system looks like:

In the Excel solution on the diagram above, data is retrieved out of the corporate database through ODBC and used to populate an Excel PivotTable on a worksheet. The PivotTable gives the user the ability to view various different pieces of data in different dynamic views. After looking at the data in a numerical fashion on the PivotTable worksheet, the user can then view the data in a graphical manner through the Excel Chart Object. It is possible to link the Chart Object to the PivotTable in order to get the same dynamic views via the Chart Object that were available from the PivotTable directly. This dynamic viewing capability is achieved on a chart through the use of on-sheet controls that are linked back to the pivottable and actually serve to control the pivottable. After viewing the data in both a numerical and a graphical fashion, the user may wish to make some changes to the data - this is often the case when a user is working on, say, a budget. The user then goes into the third box of the Excel application above and is presented with a standard Excel worksheet table that again summarizes the data that was displayed in the pivottable (as with the chart object, it is possible to have this range object linked back to the pivottable with on-sheet controls). In this screen, the user has the option of changing the values in the underlying database. Visual Basic for Applications provides numerous commands that facilitate the transfer of data between Excel and ODBC-supporting external databases. After making the desired changes, the user can move on to the next part of the application, represented by the fourth box in the diagram, and send out a summary of the data to colleagues within the company over the electronic mail system via MAPI. Lastly, if the user so chooses, he/she can create a printed report that summarizes the data using Microsoft Word and transfer data to Word via the OLE Automation feature of OLE 2.0.

Applications similar to the one above are experiencing a rather high rate of proliferation throughout corporate America. Corporations are starting to realize that decision-making by

middle-level managers is greatly enhanced by their ability to analyze and interpret large amounts of corporate data while sitting at their desks. Never before has such analysis been accomplished so easily. And among other things, Excel 5.0 is, perhaps, the most powerful tool for this type of analysis.

Now that we're reviewed an example of how Excel objects can be integrated into a custom solution, let's take a look at the technology that allows for this integration.

Excel Objects have Numerous Properties and Methods

Associated with each object in Excel are various properties and methods. A property of an object is an aspect of the object that can actually take on a value or a setting. A method of an object is an action that can be performed on an object. Stepping back from the world of spreadsheets, let's take a look at a real-world example of how properties and methods work. Let's assume that you are dealing with a tree (a real tree, such as an oak tree) as an object. The tree object has numerous properties - including height, width, trunk circumference, age, color, etc. All of these properties have certain values or settings associated with them: height=20ft; width=5ft; age=2 yrs; etc. The tree object also has various methods - such as the "grow" method, or the "photosynthesize" method, or the "give-shade" method - these are all actions which are performed by the tree, or are performed on the tree.

Now let's take a look at a spreadsheet object - specifically the Range object. The Range object refers to a range on the spreadsheet itself - a cell or a group of cells. The Range has several properties including Name, Value, Formula, RowHeight, etc. All of these properties have values or settings associated with them. The Range object also has numerous methods, such as Clear (deletes the contents of the range), Copy (copies the contents of the range), Sort (sorts the values of the range), etc. All of these Range methods represent actions that can be performed on the range of data.

The process of programming Excel 5.0 objects involves assigning values or settings to the various properties of the objects and calling the various methods of the objects as well. Such manipulation of Excel objects is accomplished through use of Excel's new macro language, Visual Basic for Applications.

2. Excel 5.0 Offers an Advanced Programming Language - Visual Basic for Applications

Microsoft Excel version 5.0 will be the first application to host Microsoft's new common macro language, Visual Basic for Applications (VBA). VBA is based on Microsoft's award-winning Visual Basic Programming System - in fact, the VBA language syntax is identical to Visual Basic, with the exception of a few new language constructs introduced in VBA. Visual Basic has been described by the computer industry trade press as one of the most powerful development tools for the Windows platform, be it for quick development of prototypes or full-blown applications development. Now, with Visual Basic for Applications in Excel 5.0, developers can utilize this powerful programming language to tie together Excel objects in creating custom applications.

Spreadsheet macro languages have traditionally been either keystroke or function based. Those who have development experience with such macro languages are well aware of their drawbacks.

Keystroke-based languages tend to be lacking in power and flexibility and involve mere emulation of the keystrokes that a spreadsheet user would use to complete a task. Function-based macro languages, while being more powerful than keystroke-based ones, are often found to be difficult to work with. Functions usually contain long argument lists for which there are a multitude of values - this makes it difficult for a developer to memorize the exact syntax of a command and requires that reference information on the language always be within easy reach.

With the introduction of Visual Basic for Applications in Excel 5.0, Microsoft is offering the first object oriented macro language in an applications program - one that is based on a mainstream programming language. An object oriented language, such as VBA, provides several advantages over keystroke and function based languages. First of all, VBA is much more powerful than other macro languages, giving developers access to the properties and methods of the advanced programmable objects available in the Excel environment. Second, VBA is easy to use - because of it's object oriented architecture, VBA makes it easy to reference properties and methods of Excel objects using an English-language-like syntax. Lastly, VBA provides a powerful editing and debugging environment, giving the developer tools that are not even available in high-level, standalone programming languages. These advantages offered by VBA combined with the Excel Object Model, provide an environment for developers to create more powerful applications in less time.

Visual Basic for Applications is Powerful

VBA is, perhaps, the most powerful macro language available in an applications program. In addition to giving you easy access to properties and methods of all Excel objects, VBA provides all of the basic language constructs found in traditional programming languages, as well as new constructs not found in any other macro language. Examples of such new constructs include the For-Each construct, that provides a means for performing the same action on entire collections of like objects, and the With construct that allows you to abbreviate object references for easy setting of multiple object properties. VBA also provides full support of variables of all types (including variant and user-defined), constants, arrays, and function macros.

Keystroke-based macro languages provide a developer a mechanism for automating the tasks that a spreadsheet user would perform by selecting commands from a menu - the keystrokes in the language are actually those that would be entered by the user on the keyboard. The power, therefore, of a keystroke-based language is limited to those commands available from the menu. Visual Basic for Applications with it's object oriented architecture provides more power than a keystroke-based language in that it provides a developer with the means of manipulating and controlling the components of a spreadsheet directly without having to go through the menu structure - this results not only in a performance advantage, but a greater degree of flexibility in application design. Additionally, VBA provides the means for accessing numerous powerful components of the application that are not even available to an end-user through the menu structure.

As do object oriented languages, function-based macro languages provide a greater degree of power than do keystroke-based ones. However, there are still limitations to function-based languages. Implementation of a function-based macro language is done with the assumption that the components of the development environment can be molded to fit within the confines of the language's functions. If we look at this from the context of a spoken language, function-based languages are essentially based on a set of verbs - functions, in effect are actions that can be performed by or on the components of the spreadsheet. This imposes limitations on the power of

the language. Imagine, if you will, a spoken language with a very limited number of verbs, while at the same time, having an unlimited number of nouns. With such a language, it would be necessary to mold or fit the nouns into the requirements of the available verbs - posing limitations on the types of sentences that could be created. These same limitations are found in function-based languages, where there are a limited number of functions. With a limited number of functions, it is necessary to attempt to mold and fit the components of the spreadsheet into the confines of the functions. To accommodate this, function-based languages tend to have functions that are very general in nature, that can be applied to a large number of different spreadsheet components. This generality reduces the efficiency with which code can be written and executed. Object oriented languages, such as VBA, provide an emphasis on the spreadsheet components, or objects, and not on the actions that can be taken by or on those components. The result is an environment that can have an unlimited number of objects, each with its own unique set of properties or actions - resulting in greater efficiency with which code can be written and executed. With VBA, spreadsheet components do not have to be molded to fit the confines of functions - instead, spreadsheet objects with their own associated properties and methods can be more efficiently integrated into more powerful applications.

Visual Basic for Applications is Easy to Use

The second area that VBA provides an advantage over keystroke and function based macro languages is in ease-of-use. With VBA, Excel objects are arranged in a hierarchy in which certain objects are contained within others. Developers can easily drill down through this hierarchy by using the dot (.) operator. Additionally, developers can access properties and methods through the use of the dot operator. For example, in order set the name property of the range object, a developer would use the following line of code:

```
Application.Workbooks("Book1").Worksheets("Sheet1").Range("A1").Name =
"CellA1"
```

The above example shows how the dot operator is used to drill down through the hierarchy of objects and then reference a property of a specific object. Looking at the individual components of the example, the reference to “Application” is actually a reference to Excel, itself. The Application object is the most global object in Excel - it refers to the Excel object, and all other objects within Excel are contained within Application. The second component of the line of code above, Workbooks(“Book1”), is a reference to a Workbook object, or in this case, to a particular workbook named Book1. Workbook objects are representative of Excel files - in the command above, the reference is to the Book1 file. The third component is a reference to a worksheet within the workbook - in this case Sheet1. The next component, Range(“A1”), is a reference to a Range object (a cell or group of cells) in the workbook file - specifically, to the cell A1. While the first four components of the line of code above have actually been references to objects, the fifth component is a reference to a property - to the Name property of the Range object. Standing back and looking at the first five components, it can be seen that the first four object references are used to drill down through a hierarchy of objects to the desired object, Range, while the fifth component is used to reference a property of the desired object, Name. The last component of the line of code is used to assign a setting or a value to the property. This is done through use of the equals (=) operator, and in this case, “CellA1” is being assigned to the Name property of the particular Range object.

It is important to note that the same outcome of executing the line of code above can be accomplished through similar but abbreviated versions of such code. For example, executing the

following line of code will provide the same desired results:

Workbooks("Book1").Worksheets("Sheet1").Range("A1").Name = "CellA1"

You'll notice that the only difference between this line of code and the one above is that the reference to the Application object is missing. Provided that the code is executed within Microsoft Excel, then no reference to Application is required. The code can be abbreviated even further, as in the following example:

Range("A1").Name = "CellA1"

In this example, references to Application, the Book1 Workbook object and the Sheet1 Worksheet object are missing. Proper execution of this line of code would require that it be executed within the Excel Application while the Sheet1 Worksheet object in the Book1 Workbook object is active. This line of code cannot be abbreviated any further - it contains minimum components for setting a property - object, property and setting.

The above provides an example of object oriented programming using Visual Basic for Applications. Let's now look at examples of code used to set the name of a range using Object Oriented, Keystroke-based and Function-based macro languages:

Example 1: Object Oriented Visual Basic for Applications macro language in Excel 5.0:

Range("A1").Name = "CellA1"

Example 2: Keystroke-based macro language in Lotus 1-2-3 for DOS and Lotus 1-2-3 for Windows:

/rnccella1~

Example 3: Function-based XLM macro language in Excel 4.0:

=DEFINE.NAME("cella1","=Sheet1!R1C1")

The first example shows Visual Basic for Applications. Now, we have already looked at this line of code in some detail, however, you will notice that the syntax and the use of the dot operator in VBA make this code very easy to read and understand - it is quite obvious that the code is being used to set the name of the range.

The second example shows how the same result would be accomplished using the keystroke-based macro language in Lotus 1-2-3 for DOS and Lotus 1-2-3 for Windows. If you are not familiar with Lotus 1-2-3, the code may appear a bit cryptic. It actually represents a set of keystrokes that a user would issue from the keyboard to execute menu commands to set the name of a range. The following breaks down the line of code with an explanation of the keystrokes:

/ - character to access 1-2-3 main menu
r - selects range entry in 1-2-3 main menu
c - selects create entry in 1-2-3 range menu
cella1 - name of range entered
~ - Enter key on the keyboard

The code example showing Lotus 1-2-3's keystroke-based macro language is actually much more concise than that for VBA. However, it is very difficult to read - and write, for that matter. In fact, even those developers who are accustomed to Lotus 1-2-3 have to break down and step through the components of keystroke-based commands in order to understand them. The advantage that VBA offers over keystroke-based languages is that VBA provides an English language-like syntax that is easy learn, write, read and understand. Instead of having to memorize a series of menu commands in order to enter the correct sequence of keystrokes, with VBA, you only need to know the names of objects, the properties and methods associated with them and the values or settings that can be assigned to them.

The third example above shows how to set the name of a range using Excel's older XLM macro language (also known as the Excel 4.0 macro language). It employs the DEFINE.NAME function and requires the passing of two arguments - the name and the address of the range. As with the 1-2-3 keystroke example, this function-based XLM example is rather cryptic and difficult to understand. Mastering function based languages requires that developers become intimately familiar with the different arguments that a function accepts, the proper ordering of those arguments and the required syntax for passing values for such arguments. The above example of a function-based command is actually quite simple - most function-based languages tend to have very complicated functions that have numerous arguments, making it very difficult for a developer to remember the names, ordering and proper syntax for the arguments. In contrast, VBA is much easier. A developer need only remember the name of the object and the property to be set. This not only makes VBA code easier to read and understand, but it is much easier to write as well.

Visual Basic for Applications Provides Advanced Editing and Debugging Tools

Visual Basic for Applications in Excel 5.0 is more than just a macro language. It is, in fact, an advanced software development environment with a complete set of editing and debugging tools. When creating VBA macros in Excel 5, VBA code is actually written on separate VBA modules, similar to code modules in Visual Basic 3.0 and other programming languages. This marks a much-welcomed deviation from the traditional practice of storing macro code on sheets in spreadsheet programs. Worksheets - the actual sheets used for storing and manipulating numerical data in spreadsheet programs - are not at all well-suited for storing and editing code. Seeing the great advantage separate code modules offer to developers, Microsoft has made Excel 5.0 the first spreadsheet program to provide separate editing modules for macro code.

Within a VBA module in Excel 5.0, there are several editing and debugging features which

cannot be found in any other spreadsheet program:

Syntax Checking: VBA actually checks the syntax of your code as you type it in. VBA automatically capitalizes the first letter of all recognized objects, properties and methods giving the developer visual feedback that object references have been made correctly. VBA also gives the developer the option of having error messages displayed automatically if major errors in syntax occur.

Color coding: VBA allows you to establish different color fonts in your code. For example, standard code can be colored in blue while erroneous code is colored in red and commented code is colored in green to make such code stand out.

Break Points: VBA provides a mechanism for setting breakpoints in your macros. While executing a VBA macro, when a breakpoint is encountered, Excel halts execution and automatically displays VBA's Debugging Window - providing a tool for a developer to locate the source of bugs.

Watch Variables: VBA provides a mechanism for establishing certain variables or statements with VBA code as Watch Variables. The value of a watch variable can be viewed at any time using the Debug Window. So for example, if a developer has isolated the source of a bug to a few variables, and the developer would like to determine the value of such variables at a particular place in the macro, a breakpoint could be set and then, upon execution of the code, when Excel displays the Debug Window, the developer can view the values of the Watch Variables to help isolate the bug.

Debug Window: As mentioned above, the debug window is very useful in isolating bugs in your code. The debug window can be brought up at any time by pressing the F8 key or by clicking on the Step Macro button on the Visual Basic toolbar. Within the Debug Window, there are actually three separate panes - the Code Pane, the Watch Pane and the Immediate Pane. The Code Pane displays the VBA macro code as it is being executed. The Watch Pane displays all of the designated watch variables and their associated values. The Immediate Pane allows the developer to execute code statements outside of the body of the macro - this is very useful for testing different assumptions on values for different variables.

Object Browser: The Object Browser is, perhaps, one of the most powerful tools available in the VBA editing environment, providing a means for listing and getting information on all of the programmable objects and their associated properties and methods available on the system. Such objects displayed by the Object Browser could exist not only within Excel itself but in other OLE 2.0 applications on the system. For example, if a developer has both Microsoft Excel 5.0 and Microsoft Project 4.0 (both of which support OLE 2.0 and have VBA) installed on his/her system, the Object Browser will display all of the objects in both Excel and Project. The Object Browser provides a window to the richest set of programmable objects ever available in a development environment. The Object Browser can be accessed by selecting the Object Browser button on the Visual Basic toolbar or by pressing F2 in a VBA module.

VBA Help: Visual Basic for Applications comes with full on-line help in Excel. Information is provided on all objects, properties, methods, functions and keywords in VBA. Help can be accessed through the Help menu or by selecting a particular object, property or method and pressing the F1 key - this will cause the help screen for the selected object, property or method to be displayed.

Additionally, Excel 5.0 offers a macro recorder which will actually write macro code for you. You merely turn on the recorder, perform the action that you would like code generated for and then turn off the recorder - Excel will create a VBA module and insert the code that corresponds to the actions taken. Excel's macro recorder serves as a very useful learning tool for those just getting started on VBA.

As can be seen from the list of features above, VBA provides an advanced editing and debugging environment - one that is on par with other mainstream development environments but greatly surpasses those environments which are found in other spreadsheet programs.

3. Excel 5.0 Provides Support for Object Linking and Embedding (OLE) ver. 2.0.

Microsoft Excel 5.0 provides full support for the Visual Editing, Drag & Drop Across Applications and OLE Automation features of OLE 2.0. No other spreadsheet program provides this level of OLE support. Of prime importance to developers is Excel 5.0's support for OLE Automation. Because Excel 5.0 supports OLE Automation, it is very easy for developers to integrate Excel with other OLE 2.0 supporting applications into custom solutions. Prior to OLE 2.0, integration of multiple applications into a single custom solution could only be accomplished through a technology known as Dynamic Data Exchange (DDE). DDE proved very difficult to work with as it required execution of numerous commands - commands for establishing DDE channels and then passing commands from one application to another. DDE was also plagued by slow performance.

OLE Automation represents a major advance in cross-application programmability. While not only demonstrating a substantial improvement in performance, cross-application development via OLE Automation is also easier and more efficient than with DDE. With OLE Automation, instead of sending commands over channels (as was the case of DDE), a developer can access another application's objects as though they were native to the application the developer is working in. For example, because Excel 5.0 and Microsoft Project 4.0 both support OLE Automation, a developer can write a VBA macro in Excel 5.0 that accesses Microsoft Project's objects - and actually sets the properties of and calls the methods of such objects as though they were native to Excel. Such an advance in cross-application development will result in developers being able to easily integrate the objects available in multiple applications in creating a single custom solution - allowing developers to create more powerful solutions with less effort.

4. Excel 5.0 Offers Support for WOSA Data Transfer Protocols - ODBC, MAPI and TAPI.

Microsoft Excel 5.0 provides support for Windows Open Services Architecture (WOSA) data transfer protocols, including Open Database Connectivity (ODBC), Microsoft's Messaging Applications Programming Interface (MAPI) and Microsoft's Telephony Applications Programming Interface. Let's look at Excel's support for each of these protocols in greater detail:

ODBC: Excel provides a means for getting data in and out of all ODBC-supporting external databases. In fact, Excel 5.0 ships with ODBC drivers for a number of popular database formats including dBase, Fox, Access, Oracle and DEC RdB. Other ODBC drivers are available through Microsoft add-on paks or from database vendors directly. Excel provides two different methods of exporting data to or importing data from external databases - through using Microsoft Query or through using SQL VBA commands. Microsoft Query, which is shipped with Excel 5.0, is more suited for end-users and provides an intuitive user interface for making external database queries. Microsoft Query is based on the external database front-end in Microsoft Access, and is compatible with both Microsoft Excel and Microsoft Word. Excel's SQL VBA commands allow a developer to issue any number of SQL statements in executing from basic to complex queries. These SQL VBA commands require the use of an Excel addin, XLODBC.XLA, which is shipped and installed with Excel 5.0. The SQL VBA commands can be used to pull data directly into or export data directly from an Excel worksheet or VBA array.

MAPI: Microsoft Excel 5.0 offers extensive support for MAPI, making it possible for developers to integrate Excel applications with electronic messaging functions available through MAPI-supporting email systems. With MAPI support in Excel 5.0, it's possible to route worksheets, access email address books, specify message recipients, package worksheets in email messages, and enter text messages. It's also possible to search for and retrieve messages out of a user's inbox. Use of MAPI functions requires the Excel 5.0 version of the XLMAPI.XLL addin available at no charge from Microsoft Customer Service.

TAPI: Excel 5.0's support for TAPI allows developers to integrate real-time data feed via modem directly into Excel applications. With support for the Telephony API, Excel 5.0 is well-suited for accepting data from and transferring data to remote locations.

5. Excel is the Market-Leading Spreadsheet Across All Platforms

The last, but nonetheless, very important reason for developing custom applications in Excel 5.0 is that Excel currently is the world's best selling spreadsheet with market leadership across all platforms - Windows, Macintosh and DOS combined. Excel currently has over 7 million installed users worldwide. Both of these statistics support the fact that there is a potentially very large market of customers for custom Excel applications. Any developer looking for a market for a custom Excel application won't have far to go - any Fortune 500 company is likely to have a large installed base of Excel users. When it comes down to it, if you're going to create a custom application in a spreadsheet, Excel is your best choice, not only in terms of power and ease-of-use, but in marketing potential as well.

III. The Excel Object Model

Microsoft Excel 5.0 contains 128 programmable objects that are arranged in a hierarchical order. Associated with each object are numerous properties and methods. A developer integrates Excel's objects into custom applications by using Visual Basic for Applications to assign values or settings to the properties of or by calling the methods of various objects. Excel 5.0 provides the most powerful object model ever made available in a development environment, giving third party developers access to the code routines and algorithms of the world's best selling spreadsheet program for both the Windows and Macintosh platforms.

1. VBA Basics

Before actually taking a look at some of Excel's objects, it's important to have a good understanding of some of the basics behind Visual Basic for Applications. As stated above, VBA macros are stored on separate code modules within Microsoft Excel 5.0. A VBA code module is inserted into an Excel 5.0 workbook by selecting the Macro command from the Insert menu and then selecting Module.

There are two types of VBA macros - procedures and functions. The main difference between procedure macros and function macros is that function macros return values while procedure macros do not. Our main focus is going to be on procedure macros. All procedure macros begin with the keyword Sub and end with the keywords End Sub. The following is an example of a VBA macro that displays a message box:

```
Sub Say_Hello()
   MsgBox "Hello"
End Sub
```

Note that at the beginning of each VBA procedure macro after the Sub keyword is the actual name of the macro - the macro above is named Say_Hello. You run VBA macros in Excel by selecting the Macro command from the Tools menu and then selecting the name of the macro you wish to run. It is important to note that there are several other ways of executing VBA macros within Excel - these different methods of execution are also known as "Events". Perhaps, the most common event for macro execution in Excel is the "Click" event - this is accomplished by assigning a VBA macro to a button on the worksheet that the user can click. This is done by first placing a button on the sheet (by selecting the Create Button tool on the Forms toolbar), clicking the right mouse button on the button, and selecting Assign Macro to Object from the shortcut menu.

Another important aspect of VBA that is important to understand is variable use and variable declarations. VBA offers a new feature that does not force you to declare variables before they are used. For example, the following VBA macro assigns a value of 1 to an integer variable named int_one, even though int_one is not declared beforehand. The macro then puts the value in int_one into the range A1 on the active worksheet:

```
Sub Assign_Value()
      int_one = 1
      Range("A1").Value = int_one
```

End Sub

Many development teams, however, would prefer an environment in which they were forced to make variable declarations in advance - such declarations lead to greater organization in code and often eliminate errors and bugs by misuse of variables. Forced variable declaration can be established in a VBA solution in two different ways: by selecting the Options command from the Tools menu in a VBA Module and then selecting "Require Variable Declaration" from the Module General Tab or by placing the statement "Option Explicit" at the top of a VBA module.

A variable is declared in VBA by using the Dim keyword (stands for Dimension) followed by the name of the variable and then the variable type. Going back to the macro example above, if Forced Variable Declaration were in force for the particular application, the macro would have to be modified to include a Dim statement for the int_one variable:

```
Sub Assign_Value()
    Dim int_one As Integer
    int_one = 1
    Range("A1").Value = int_one
End Sub
```

Variables can take on several different data types in VBA including:

Boolean (2 bytes)
Integer (2 bytes)
Long (4 bytes)
Single (4 bytes)
Double (8 bytes)
Currency (8 bytes)
Date (8 bytes)
Object (4 bytes)
String (1 byte per character)
Variant (16 bytes + 1 byte per character)
User-defined (number of bytes required by element)

Additionally, you can use variables to represent arrays. An array variable declaration is accomplished by specifying the name of the variable along with the dimensions of the array in parenthesis as the following example shows:

```
Dim int_numbers(10,10) As Integer
```

The statement above declares a two dimensional 10x10 array of integers. An array can have up to 60 dimensions.

A very common use of variables in VBA code is to determine the setting of a particular object property. For example, if you wanted to display the contents of Range("A1") in a message box through use of a variable:

```
Sub Display_A1()
    Dim int_one As Integer
    int_one = Range("A1").Value
    MsgBox int_one
End Sub
```

Three very powerful data types in Excel are the Object, Variant and User-Defined data types. Object variables are extremely useful in that they can be used to represent Excel objects. In order to assign an Excel object to an object variable, you must use the Set keyword. The following example shows how an object variable can be used to assign a value ("Hello") to the range A1 on worksheet Sheet1 in workbook file Book1.xls:

```
Sub Assign_A1()
    Dim cella1 As Object
    Set cella1 = Workbooks("Book1.xls").Worksheets("Sheet1").Range("A1")
    cella1.Value = "Hello"
End Sub
```

The variant data type allows you to declare and assign values to variables without knowing in advance the data type of the values being assigned. For example, let's assume that there is a value in cell A1, but it is uncertain what the data type of the value is. The following code could be used to display the value in a message box:

```
Sub Get_A1()
    Dim cella1
    cella1 = Range("A1").Value
    MsgBox cella1
End Sub
```

Notice that no data type is specified in the variable declaration of cella1 in the example above - this causes VBA to default to a variant data type.

A User-Defined data type must be declared outside of a procedure by using the Type keyword - in fact, such a declaration must occur at the very top of a VBA module before any procedures. The following example shows declaration of a user-defined data type and then use of a variable that conforms to the data type to display data in a message box:

```
Type MyData
    MyName As String
    MyAge As Integer
End Type

Sub Person_Data()
    Dim Person_x As MyData
    Person_x.MyName = "John"
    Person_x.MyAge = 23
    MsgBox Person_x.MyName & ", " & Person_x.MyAge
End Sub
```

Variables can have different levels of scope in VBA. Variables declared within a procedure have a procedure-level scope in that they can only be used within the procedure. Variables declared at the top of a VBA module before any procedures have module-level scope and can be used by any procedure within the module. Module-level variables that are declared with the Public keyword (instead of Dim) have project-level (or application-level) scope in that they can be used in any VBA module. The following is an example of a Public variable declaration:

```
Public var_one As Integer
```

VBA also offers several control structures, including If-Then-Else, Select Case, For-Next and Do-While-Loop. These structures operate in VBA in basically the same way they do in other programming languages. Let's take a look at a few examples:

If-Then-Else: The following macro employs an If-Then-Else construct to test the contents of Range("A1"). If the contents equal 1, then it displays a message box that states the answer is correct. If the contents do not equal 1, it displays a message box encouraging the user to try again:

```
Sub Test_Answer()
    Dim var_one As Integer
    var_one = Range("A1").Value
    If var_one = 1 Then
        MsgBox "Congratulations!  The answer is correct."
    Else
        MsgBox "I'm sorry.  The answer is not correct.  Please try again."
    End If
End Sub
```

Select Case: The Select Case construct provides a mechanism for performing different actions based on the value of a variable. The following macro uses the InputBox function to request the user to enter a password and based on the password entered, displays a message that grants the users read-write access, read-only access or no access to the application (incidentally, the two parameters of the InputBox function correspond to prompt and title):

```
Sub Enter_Password()
   PassWord = InputBox("Enter Password:", "Password")
   Select Case PassWord
   Case "levelone"
      MsgBox "You have read-write access to the application."
   Case "leveltwo"
      MsgBox "You have read-only access to the application."
   Case Else
      MsgBox "I'm sorry, the password you have entered is not correct."
   End Select
End Sub
```

For-Next: The For-Next construct can be used to repeat an action for a certain number of times. For example, the following macro reads a value out of cell A1. It then uses a For-Next loop to calculate the factorial of the number and returns the factorial in a message box:

```
Sub Find_Factorial()
   Dim var_one As Integer
   Dim var_two As Double
   var_one = Range("A1").Value
   var_two = 1
   For x = var_one To 1 Step -1
      var_two = var_two * x
   Next
   MsgBox "The factorial of " & var_one & " is " & var_two
End Sub
```

Do-While-Loop: Do-While-Loops are very similar to For-Next loops, except instead of the loop executing a specified number of times (as with For-Next), the loop will execute until the specified condition evaluates to True. The macro below displays an input box that prompts the user to enter the name of a state. The macro will continue to display the input box until the user enters "California":

```
Sub geo_quiz()
   Dim state_name As String
   state_name = ""
   Do While state_name <> "California"
      state_name = InputBox("Enter the state name:", "Geography Quiz")
   Loop
End Sub
```

One last VBA construct that it is important you be familiar with is the With construct. The With construct provides a very useful tool in that it allows you to abbreviate object references in setting multiple properties of an object. Let's assume you wish to write a macro that sets various properties of the font of range A1 - specifically, you'd like to set the font size to 12, the font name to Times, as well as make the font bold and italic. Here's what the macro would look like without the With construct:

```
Sub Set_Font()
   Worksheets("sheet1").Range("a1").Font.Size = 12
   Worksheets("sheet1").Range("a1").Font.Name = "Times"
   Worksheets("sheet1").Range("a1").Font.Bold = True
   Worksheets("sheet1").Range("a1").Font.Italic = True
End Sub
```

Now, here is a different form of the same macro that employs the With construct:

```
Sub Set_Font()
   With Worksheets("sheet1").Range("a1").Font
      .Size = 12
      .Name = "Times"
      .Bold = True
      .Italic = True
   End With
End Sub
```

As you can see in the above example, the With construct allows you to abbreviate object references when setting multiple properties of the same object. This makes it much easier to write code, as well as to read and understand code.

Two additional points concerning VBA that it is important to note is the process of calling one VBA macro from another and the process of passing arguments between VBA macros. To call one macro from another, you merely include the name of the macro to be called as a command, as shown in the following two macros. In this example, the first macro calls the second macro:

```
Sub Macro_One()
   Do While (InputBox("Enter State:", "Geography Quiz") <> "California")
   Loop
   Display_Message
End Sub

Sub Display_Message()
   MsgBox "Your answer is correct!"
End Sub
```

The above macro uses a Do-While-Loop to continually prompt the user to enter the name of a state. If the user enters California, the macro will break out of the loop and call the Display_Message macro which will display a message box indicating to the user that the correct answer has been entered. You will notice the use of the not-equals (<>) operator used to compare the InputBox entry to the string "California" in Macro_One.

The next example displays a message box prompting the user for input and then passes the users input to the Display_Answer macro:

```
Sub Macro_Two()
   Dim State_Name As String
   State_Name = InputBox("Enter State:", "GeoQuiz")
   Display_Answer (State_Name)
End Sub

Sub Display_Answer(State_Name)
   MsgBox "Your answer is " & State_Name & "."
End Sub
```

Macro_Two above uses the variable State_Name to store the string entered using the InputBox and then passes this string to the Display_Answer macro which displays it with a message in a message box. You will notice the use of the concatenation operator (&) to concatenate State_Name with the rest of the string in the message box.

In summary, VBA provides the tools for working with Excel objects - it provides a way to access and set the values of Excel object properties and a means for calling Excel object methods.

2. Excel Objects

Now that we have some of the VBA basics out of the way, let's take a look at some Excel objects. As has already been stated in this paper, Excel objects are arranged according to a hierarchy. The most global object is the Application object, or the object which represents Excel itself. On the next two pages, there is a diagram of the Excel Object Model representing the vast majority of objects in Excel. The diagram shows the hierarchical nature of Excel objects.

It is important to note that there are many objects within Excel that are also known as Collections. Collections are merely groups of like objects. For example the Workbook object is also a collection - the Workbook Collection represents all of the open workbook files in Excel. Collections are very powerful in that they give developers the ability to perform the same action on all of the objects that fall within the collection. As an example, let's say you are interested in writing a macro that will close all of the workbook files which are currently open in Excel. In the past, if you wanted to perform this task in any spreadsheet program, you would have to write a macro that would first determine how many files were open as well as the names of the files - you would then have to implement a looping structure that would go through and close each file individually. Because of Collections in Excel 5, the task of performing the same action on multiple instances of like-objects, such as closing all of the open workbooks, is easy. Here is an example of an Excel 5.0 macro that uses the new For-Each construct to go through the collection of workbooks, closing each workbook:

```
Sub Close_Workbooks()
    For Each Wrk_book in Application.Workbooks
        Wrk_book.close
    Next
End Sub
```

As demonstrated in the example above, Collections are a very powerful tool that allow developers to implement advanced routines with fewer lines of code. In the Excel Object Model diagram that follows on the next two pages, those objects that are also collections are shaded in gray. You will also notice in the diagram, the core of the Excel model is displayed on the first page, while a diagram displaying those objects that fall under the DrawingObjects object are displayed on the second page - the DrawingObjects object is a sub-object of the Worksheet, Chart and DialogSheet objects.

As stated before, when writing VBA macros in Excel 5, you drill down through the hierarchy of objects by using the dot (.) operator. For example, a macro to insert the string “Hello World” in a cell range on a worksheet would look like this:

```
Sub Say_Hello()
   Application.Workbook.Worksheet.Range.Value = "Hello World."
End Sub
```

The user of the dot operator above shows how to navigate through the hierarchy of objects. Unfortunately, if you were to attempt to run the above macro in Excel 5.0, it would not execute. This is the case because two of the objects in the hierarchy above (Workbook and Worksheet) represent collections - and when referencing specific objects within a collection, it is necessary to specify the exact name of the object that is to be referenced. This is also the case with the Range object, although Range is not itself, a collection. For a collection, you specify a specific object by enclosing the quoted name of the object in parenthesis after the name of the collection. In other words, we can’t merely tell Excel to place the string “Hello World” in “Range” in “Worksheet” in “Workbook”, we have to tell Excel which range in which worksheet in which workbook. Making these adjustments, the macro written with proper syntax would look like this:

```
Sub Say_Hello()
    Application.Workbooks("Book1.xls").Worksheets("Sheet1").Range("A1").Value = "Hello World."
End Sub
```

Now as stated before, if you are running this code within Excel 5.0, you don't really need the reference to Application, because Application just represents the Excel 5.0 object - however, a reference to Application would be required if you were running this macro from outside of Excel, in Visual Basic 3.0 or Microsoft Project 4.0, for example. Also note that collections allow you to reference an active object without actually calling it by name. For example, if I know that the only workbook that is currently open in Excel 5.0 is Book1.xls, then the Workbooks reference above could be changed to ActiveWorkbook, as the following code example demonstrates:

```
Sub Say_Hello()
    ActiveWorkbook.Worksheets("Sheet1").Range("A1").Value = "Hello World."
End Sub
```

Likewise, if Sheet1 is the active worksheet, Worksheets reference can be changed to ActiveSheet:

```
Sub Say_Hello()
    ActiveWorkbook.ActiveSheet.Range("A1").Value = "Hello World"
End Sub
```

But, if Sheet1 in Book1.xls is active, this code could be abbreviated even further to:

```
Sub Say_Hello()
    Range("A1").Value = "Hello World"
End Sub
```

And if Range A1 has already been selected, it can be abbreviated further to:

```
Sub Say_Hello()
    Selection.Value = "Hello World"
End Sub
```

So, essentially, all of the lines of code below perform the same action (assuming the right workbook, worksheet and range is selected):

```
Application.Workbooks("Book1.xls").Worksheets("Sheet1").Range("A1").Value = "Hello World."
ActiveWorkbook.Worksheets("Sheet1").Range("A1").Value = "Hello World."
ActiveWorkbook.ActiveSheet.Range("A1").Value = "Hello World"
ActiveSheet.Range("A1").Value = "Hello World"
Range("A1").Value = "Hello World"
Selection.Value = "Hello World"
```

In contrast to properties, object methods do not take on values, rather they cause an action to be performed on a particular object. Methods do, however, take arguments (both required and optional) that can be used to indicate how such actions are performed. A good example is the Close method of the Workbook object. The Close method is used to close an Excel worksbook and can take three arguments:

SaveChanges - if True, the workbook file will be saved before closing.
FileName - the name of the file the workbook will be saved to.
RouteWorkbook - if true, the workbook will be routed to the next user on the routing list.

All three of the above arguments are optional for the Close method, so it would be possible to call the Close method as in the following example:

```
Workbooks("Book1.xls").Close
```

The above command would cause Excel to automatically prompt the user to save changes. The Close method could also be called by passing the optional paramters. In the example that follows, the Close method is called in such a way that the file is saved to the file "Book2.xls" prior to closing, and the workbook is not routed to the next user:

```
Workbooks("book1.xls").Close savechanges:=True, filename:="Book2.xls", routeworkbook:=False
```

You'll notice in the example above, that values are assigned to the method arguments by using the colon-equals (:=) operator. It is possible to pass values for these arguments without referencing the argument names provided that the values are passed in the right order (as listed above). Using the argument name with the colon-equals operator allows you to selectively pass values to arguments - you don't have to pass values to all arugments - and it also allows you to pass arguments in any order. For example, the following line of code closes the Book1.xls workbook in exactly the same manner as the line of code above, however, you will notice that the arguments are in a completely different order:

Workbooks("book1.xls").Close filename:="Book2.xls", routeworkbook:=False, savechanges:=True

If you pass values to arguments in a method but do not include the names of the arguments, you must make sure that the values are passed in the correct order. The following line of code shows an example of closing a workbook in the same way as above but without including the argument names when passing the values to the arguments:

Workbooks("book1.xls").Close True, "book2.xls", False

The above examples show the power of VBA in assigning values to the properties of and calling the methods of objects in Excel 5.0. Now that we have a sense of the hierarchy of objects, let's take a look at some of Excel 5.0's objects and the properties and methods associated with them.

The Application Object

As has already been stated, Application refers to the Excel Object, the most global object in Excel 5.0 - the object at the top of the hierarchy. Changes made to the properties of the Application object, as well as methods called on Application, affect global settings for the entire environment. The Application Object is also used to reference Excel 5.0 from an external application. For example, a macro can be written in Microsoft Project 4.0 that references the Application Object in Excel through OLE Automation - this will allow Project to execute Excel VBA macro commands from outside of Excel.

From a development standpoint, the Application Object should be viewed as the environment under which an Excel VBA custom solution will run. Therefore, changes made to the properties of the Application Object and methods run on the object will affect the environment in which the solution runs, not the solution itself.

The Application Object has over a hundred properties and methods. The best way to get information on the properties and methods of any object in Excel is to use either the on-line VBA help shipped with the product or to access the Object Browser in a VBA module. To access VBA Help, select Contents from the Help Menu (or press F1) and then select Programming with Visual Basic from the Help Contents screen. Using VBA Help, you can actually access a list of all the objects in Excel 5.0 and by selecting an object from the list, display all of the properties and methods for the selected object. You can drill down further and get specific information on each property and method, and even get code examples that you can copy and paste into a VBA module in your Excel application. To access Excel's Object Browser, select the Object Browser tool on the Visual Basic toolbar or press the F2 key in a VBA module.

Object properties in Excel can only take on one specific value at any one time. Methods, as you may recall, don't have values associated with them - that's because methods represent actions performed by or on an Object. Methods, however, can have optional parameters that specify how the method or action is to be carried out. For example, the Range object has a method called AutoFormat. You can call the AutoFormat method without any parameters and Excel will apply the last selected autoformat style to the specified range. However, Excel has many built-in autoformat styles - because of this, it is possible to pass the name of an autoformat style to the AutoFormat method, causing Excel to implement the style of your choice. This is accomplished by specifying the name of the parameter and then using the colon-equals (:=) operator to assign a value to the parameter, as the following example shows:

Range("A1:D5").AutoFormat Format:=xlClassic3

The above line of code applies the xlClassic3 autoformat style to the range A1:D5.

There's certainly not enough room here to discuss all of the properties and methods associated with the Application object, however, we will take a look at a few of them.

Application Properties

ScreenUpdating

One of the most useful properties of the Application Object is the ScreenUpdating property. This property can take on one of two values - True or False. It is equivalent to the popular "Echo On" or "Echo Off" found in other programming languages, and has the effect of allowing a developer to have control over what actions performed in the spreadsheet the user will actually see. The following line of code sets the value of the ScreenUpdating property to False, thereby, causing subsequent actions performed in the spreadsheet by a VBA macro to be removed from the view of the user:

Application.ScreenUpdating = False

Caption

Another very useful property of the Application object is the Caption property - the Caption setting takes a string value and controls what is displayed in Excel's title bar. The following example customizes the title of Excel:

Application.Caption = "Nancy's Custom Excel Application"

The default setting for the Caption property is "Microsoft Excel" and you can reset the Caption back to "Microsoft Excel" by assigning the constant Empty to the property:

Application.Caption = Empty

DiplayAlerts

Another useful property of the Application object is the DisplayAlerts property - this will allow the developer to control whether Excel's built-in alerts are displayed to the user. DisplayAlerts can take on values of True or False - the following example turns off DisplayAlerts:

Application.DisplayAlerts = False

StatusBar
The StatusBar property takes a string value and controls what is displayed in the Status Bar at the bottom of the Excel screen:

Application.StatusBar = "You are running a custom application in Excel."

Application Methods

Calculate
The Calculate method forces Excel to recalculate all formulas on all worksheets in all open workbooks. The Calculate method does not take any optional parameters and can be called as in the following example:

Application.Calculate

Help
The Help method will display a help topic from a windows help file. It takes two optional parameters, the name of the help file and the context ID of the help topic. If the help topic context ID is not included, the contents window for the help file will be displayed. The following example displays the help contents windows for the file Custom.hlp:

Application.Help helpfile:="Custom.hlp"

Quit
The Quit method has the effect of exiting Excel:

Application.Quit

Run
The Run method can be used to run Excel XLM macros from within a VBA module. For example, if you have an XLM macro named Macro_one, you could call it from a VBA module by using the Run method:

Application.Run "Macro_one"

The Workbook Object

The Workbook Object represents the Excel Workbook File. It is important to note that workbook files can contain different types of sheets: worksheets, dialogsheets, charts, VBA modules and XLM macro sheets. A single workbook file can have an unlimited number of sheets - this represents a vast improvement over the traditional one-sheet per one-file spreadsheet paradigm.

In looking at Excel 5.0 as a development platform, the Workbook Object is used to hold the components of a custom VBA solution. While it is possible to integrate multiple workbooks into a single VBA solution, it is more often the case that a single workbook holds a single solution. Whereas property settings and method calls made on the Application object will have no affect on the custom solution itself (but rather, on the environment), property settings and method calls on the Workbook Object will actually be acting on the custom solution. As with the Application Object, the Workbook Object has hundreds of properties and methods - too many to go through in this document. Before looking at some code examples, recall that the Workbook Object is also a Collection. Because Workbook is a Collection, you can reference individual workbook files by passing their quoted names in parenthesis, or you can reference the active workbook by using ActiveWorkbook. Let's take a look at just a few examples.

Workbook Properties

Name

The Name property refers to the actual file name of the workbook. Assuming that the active workbook has not yet been saved to a filename, you could assign one using the following:

```
ActiveWorkbook.Name = "Custom.xls"
```

Path

Assuming then, that you did save the workbook file to the file name above, you could determine the DOS path by using the following VBA macro:

```
Sub Find_Path()
    Dim path_name As String
    path_name = Workbooks("Custom.xls").Path
End Sub
```

Author

The Author property can be used to retrieve or to set the name of the author of the workbook as a string. This is useful for tracking who the original creator of a workbook is or for keeping track of who was the last to make changes to a workbook:

Workbooks("Custom.xls").Author = "John"

Workbook Methods

Activate
The Activate method will activate the specified workbook displaying the first window associated with the workbook. For example, if you have several workbooks open in Excel and you would like to activate Book1.xls, you would use the following code:

Workbooks("Book1.xls").Activate

Close
The Close method will close a workbook:

Workbooks("Book1.xls).Close

Workbooks.Open
If you wish to open an existing workbook, you must use a different object - that is the Workbooks object. Notice the only difference in syntax between the Workbook and the Workbooks object is that one is plural and one is not. Workbooks is an object which actually refers to the Workbook Collection. So, by using the Workbooks object, you are, in effect, adding another workbook to the Workbook Collection. Here's an example of how it's done:

Workbooks.Open FileName:="Book2.xls"

Workbooks.Add
As in the example above, if you wish to add a new workbook, you must use the Add method of the Workbooks object;

Workbooks.Add

Save
The Save method saves the workbook:

Workbooks("Book1.xls").Save

The Worksheet Object

In terms of development, the Worksheet Object in Excel 5.0 serves as the basis for "forms" in your custom solution - it represents the main interface that a user will interact with. Worksheets lie at the heart of all Excel solutions. You'll find that the vast majority of time spent by users of custom Excel applications is spent manipulating data or other Excel objects on a worksheet. The Worksheet object contains more sub-objects than any other of Excel's main objects and you'll find worksheets to be extremely flexible and powerful.

A worksheet can serve a multitude of purposes:

1. User Interface: Worksheet Forms: Worksheets are extremely flexible in the way they can be customized. While the base of a worksheet is a grid of cells, worksheets can be altered in numerous ways to create custom forms. For example, the cells themselves that make up the worksheet can be treated as objects (Range objects) that have several formatting properties associated with them - font, color, border, style, width, height, etc. Additionally, it is very easy to add built-in or custom graphic objects to a worksheet for advanced form design. Excel also has a worksheet customization feature not available in other spreadsheet programs - on-sheet controls. With on-sheet controls, you can actually place controls such as dropdown list boxes, optionbuttons, checkboxes, scrollbars, and even multi-select listboxes directly on a worksheet. All of these worksheet customization features allow for advanced forms design within the Excel environment. And because Excel workbook files contain multiple worksheets, it's very easy to design multiple forms for your custom solutions.

2. Data Display: Because of the grid-like nature of the worksheet, it is ideally suited for displaying data tables. If your custom solution requires the display of numerical data in easy to read tables, then Excel worksheets provide a mechanism for such display. And because of Excel's numerous formatting features, including it's AutoFormat feature, it is rather easy to format worksheet ranges into easy-to-read, visually desirable tables. But aside from viewing data in standard Excel tables on a worksheet, Excel also provides one of the most powerful data displaying tools available in any spreadsheet, not to mention any commercially available applications program - the PivotTable. The PivotTable, as a sub-object of the Worksheet object, provides an interactive table that allows users to view data from large databases in different dynamic views. With PivotTables, a user can actually re-orient, or "pivot" data by dragging database field labels with the mouse. For example, imagine, if you will, a pivottable in which products are displayed going down the rows and months are displayed going across the columns. If a user wishes to pivot this table so that months were displayed down the rows and products across the columns, the user would only have to drag the label for months to the row area and the label for products to the column area - a truly dynamic table that provides a very easy-to-use, but very powerful interface. And because the PivotTable is an object in Excel, a developer can obtain full control over the properties and methods associated with the PivotTable by using VBA.

3. Data Manipulation: Excel worksheets come with over 400 built-in functions that span numerous categories, including financial, engineering, mathematical, statistical, logical, date/time, etc. You can add to Excel's built-in suite of worksheet functions by either writing your own function macros in VBA or by purchasing one of many commercially available add-ins for Excel. With such power behind the worksheet grid, it serves a perfect platform for performing advanced numerical computations. If your custom solution requires the computation

of the Net Present Value of an investment, don't write the code for computing the value yourself, instead, employ Excel's NPV worksheet function and have Excel do the work for you.

4. Database: Because of the grid-like structure of Excel worksheets, optimization of minor database routines can be achieved by using the worksheet as a database. With Excel worksheets having a maximum of 16,384 rows, worksheet database operations would have to be limited to 16,384 records. However, certain performance improvements are to be gained performing a query on an external database and importing records to an Excel sheet for manipulation. For small databases (with fewer than 16,384 records) you may find Excel worksheets well suited for storing database data.

Let's now take a look at some of the properties and methods associated with worksheets that allow you to achieve some of the functionality described above:

Worksheet Properties

Name
The Name property is used to set or return the name of the worksheet as displayed on the worksheet tab at the bottom of the screen:

```
ActiveWorksheet.name = "Pivot Sheet"
```

OnCalculate, OnData, OnDoubleClick, OnEntry, OnSheetActivate, OnSheetDeactivate
Worksheets have several event properties associated with them - as this list indicates. An event property can be used to set a macro that will execute whenever an event is encountered. For example, on the worksheet, it is quite common to specify a macro to run whenever data is entered - this would correspond to an OnEntry event. You only need specify the OnEntry macro once, and it is saved with the worksheet until either a different OnEntry macro is specified or until the OnEntry property is assigned to an empty string. The following two macros demonstrate first, how the OnEntry property can be set to a macro, and second, an OnEntry macro that checks if an entry on the worksheet is a number, displaying an error message if anything other than a number is entered:

```
Sub set_OnEntry()
   Worksheets("sheet1").OnEntry = "Check_for_Number"
End Sub

Sub Check_for_Number()
   If Not (IsNumeric(Selection.Value)) Then
      MsgBox "Sorry, you must enter a number."
      Selection.Value = ""
   End If
End Sub
```

To reset the OnEntry property so that no macro executes upon entry, you would use the following:

Worksheets("Sheet1").OnEntry = ""

The other Event properties operate in a similar fashion to OnEntry. OnCalculate can be used to execute a macro when ever the worksheet recalculates. OnData can be used to execute a macro whenever DDE or OLE linked data arrives in the worksheet. OnDoubleClick can be used to execute a macro whenever a user doubleclicks the mouse somewhere on the worksheet. OnSheetActivate and OnSheetDeactivate can be used to execute macros whenever the worksheet is activated or deactivated.

Previous, Next
These two properties return the names of the worksheets previous to and after the specified worksheet:

Worksheets("Sheet1").Next.Activate
Worksheets("Sheet1").Previous.Activate

Visible
The Visible property can be used to hide or unhide a worksheet. In fact, if you set the visible property to the constant xlVeryHidden, there will be no way for a user to manually unhide the worksheet:

Worksheets("sheet1").Visible = xlVeryHidden

The only way to unhide such a worksheet, would be to execute the following:

Worksheets("sheet1").Visible = True

Worksheet Methods

Activate
The Activate method activates the specified worksheet:

Worksheets("Sheet1").Activate

Calculate
The Calculate method forces the worksheet to recalc:

Worksheets("Sheet1").Calculate

Delete
Serves to delete the specified worksheet:

Worksheets("Sheet1").Delete

PivotTableWizard
The PivotTableWizard is, perhaps, one of the most powerful methods available for the worksheet object, and is used to create a pivottable. It will be discussed in great detail in the PivotTable Object section below.

Protect
The Protect method is used to protect the worksheet from any user input - different levels of protection can be established and a password can even be assigned. The Protect method takes the following four arguments:

Password - used to assign a password to the protected worksheet.
DrawingObjects - if true, on-sheet objects are protected.
Contents - if true, cell contents are protected.
Scenarios - if true, scenarios are protected (only applicable is using Scenario Manager).

The following line of code has the effect of fulling protecting worksheet sheet1 and setting a password of "password1":

Worksheets("sheet1").Protect "password1", True, True, True

Note that it is possible to selectively have different ranges or drawingobjects within a worksheet remain unprotected even after the Protect method has been called. This is done by setting the locked property of the particular range to False. The following example protects everything on the sheet except range A1:

Worksheets("sheet1").Range("a1").Locked = False
Worksheets("sheet1").Protect "password1", True, True, True

Unprotect
The Unprotect method has the effect of removing protection from everything (ranges, drawingobjects and scenarios) on the worksheet and takes as its only parameter a password (if one has been implemented in the Protect method). The following call of the Unprotect method would be used to unprotect the worksheet that was protected in the previous example:

Worksheets("sheet1").Unprotect "password1"

PivotTable Object

The Pivottable object is, perhaps, one of the most powerful objects in Excel 5.0. Pivottables provide an interactive data table that allows users to change views dynamically by dragging database labels with the mouse. Pivottables work by reading data from either an Excel worksheet or an external database into an internal memory cache. The pivottable is then constructed by placing different database fields into one of four different fields of the table: row, column, page and data fields. The row field goes down the rows of the table, while the column field goes across the columns of the field. The data field represents the body of the table itself - where the actual data is displayed. The page field provides a 3rd dimension to the table by actually providing a mechanism for changing the entire display of the table. It is possible to display multiple database fields within a single pivottable field thereby, further enhancing the dynamic views that can be achieved with pivottables. Users can manipulate the fields which are displayed in the pivottable, causing Excel to read from the pivottable cache and update the table. It is important to note that pivottables are used for viewing data only - users cannot update data in the pivottable. Also note that the pivottable cache is accessible by the pivottable only - it cannot be used for other database operations. A pivottable is created on a worksheet by using the PivotTableWizard method of the Worksheet object.

PivotTableWizard

The PivottableWizard method (of the Worksheet object) is used to create a pivottable and can take several arguments:

SourceType: Indicates whether data is coming from Excel worksheet or external database.
SourceData: The actual data source.
TableDestination: The destination of the pivottable.
TableName: The name assigned to the pivottable object that is created.
RowGrand: Iindicates if row totals are included in the table.
ColumnGrand: Indicates if column totals are included in the table.
SaveData: Indicates if pivottable cache is saved with the worksheet.
HasAutoFormat: Indicates if an Excel built-in Autoformat is applied to the table.
AutoPage: Used to created automatic page field when data source is a consolidation.

While there is not the space here to explain all of the different permutations of parameters that can be passed to the PivotTableWizard method (detailed information can be found in the VBA On-line Reference in Excel 5.0), let's look at a few examples. The first example below creates a pivottable from an Excel database that occupies the range A1:I226 on the Excel worksheet named "sheet2". The pivottable is then placed on worksheet "sheet1" starting at the range B5. Additionally the pivottable is given the name "Pivot1":

```
Sub Create_Pivottable_one()

Dim Sheet1 As Object
Set Sheet1 = Worksheets("sheet1")
Dim Range1 As Object
Set Range1 = Sheet1.Range("B5")
Sheet1.PivotTableWizard xlDatabase, Worksheets("sheet2").Range("a1:i226"), Range1, "Pivot1"

End Sub
```

The macro above employs two object variables, Sheet1, which refers to the worksheet named Sheet1, and Range1, which refers to the range B5 on worksheet Sheet1. The macro then calls the PivotTableWizard method on Sheet1 - let's look at the four parameters which are passed to the method:

xlDatabase - a constant to indicate the data is originating from an Excel worksheet database.
Worksheets("sheet2").Range("a1:i226") - the range of the Excel worksheet database.
Range1 - object variable representing cell B5 on Sheet1, where the pivottable will be placed.
"Pivot1" - the name to be assigned to the pivottable - for referencing the pivottable object.

The next example shows how the PivotTableWizard method can be called to create a pivottable from an external database source:

```
Sub Create_Pivottable_Two()

Dim Sheet1 As Object
Set Sheet1 = Worksheets("sheet1")
Dim Range1 As Object
Set Range1 = Sheet1.Range("B5")
Sheet1.PivotTableWizard xlExternal, Array("DSN=Jet3", "Select * From Jet3"), Range1, "Pivot2"

End Sub
```

This example is very similar to the first example - except for two important differences:

1. The first argument passed to the PivotTableWizard method is xlExternal. This indicates that pivottable data is going to be obtained from an external datasource. It is possible to retrieve external data from any external database that supports the Open Database Connectivity (ODBC) standard.

2. The second argument passed to the PivotTableWizard is the following:

```
Array("DSN=Jet3", "Select * From Jet3")
```

When retrieving data from an external datasource, the second argument for the PivotTableWizard method (SourceData) must be a two element array as shown above. The first element of the array is the actual Datasource Name (DSN) - in this case "Jet3". Datasource Names are specified by using Microsoft Query and are saved in the ODBC.INI file in the Windows directory. Jet3 represents an Access database (named Jet3). The second element of the array is the actual SQL

statement for querying the data source - in this case "Select * From Jet3". This SQL statement has the effect of selecting all of the records from all of the fields of the Jet3 table within the Jet3 database. While this example shows how to retrieve all of the records from an external database into a pivottable, it is possible to pass advanced SQL queries to the PivotTableWizard method for selecting portions of a database. It should be noted that Excel 5.0 limits the SQL strings that it will accept to 255 characters.

If you were to run either of the macros above, Excel would create a PivotTable on Sheet1 (provided the underlying databases were in place). However, you would not see any database field labels in the Row, Column, Page and Data fields. Further code must be written to display data in these fields. This is done by adjusting the orientation property of the PivotField object - a sub-object of the Pivottable object. Let's assume that you have the following fields in the Jet3 database from the second example above - and you would like to place these fields into the indicated pivottable fields:

Region - page field
Prod_cat - page field
Country - row field
Product - column field
Revenue - data field

The following macro could be used to actually construct the pivottable, placing the database fields into the pivottable fields indicated above by setting the orientation properties of the respective pivotfield objects. This macro could then be called by either of the PivotTableWizard example macros above.

```
Sub Construct_Pivot()

    Dim piv As Object
    Set piv = Worksheets("sheet1").PivotTables("pivot2")

    piv.PivotFields("region").Orientation = xlPageField
    piv.PivotFields("prod_cat").Orientation = xlPageField
    piv.PivotFields("country").Orientation = xlRowField
    piv.PivotFields("product").Orientation = xlColumnField
    piv.PivotFields("revenue").Orientation = xlDataField

End Sub
```

Putting a create_pivottable and construct_pivottable macro together would give you the following:

```
Sub Create_Pivottable_Two()

Dim Sheet1 As Object
Set Sheet1 = Worksheets("sheet1")
Dim Range1 As Object
Set Range1 = Sheet1.Range("B5")
Sheet1.PivotTableWizard xlExternal, Array("DSN=Jet3", "Select * From Jet3"), Range1, "Pivot2"

Dim piv As Object
Set piv = Worksheets("sheet1").PivotTables("pivot2")

piv.PivotFields("region").Orientation = xlPageField
piv.PivotFields("prod_cat").Orientation = xlPageField
piv.PivotFields("country").Orientation = xlRowField
piv.PivotFields("product").Orientation = xlColumnField
piv.PivotFields("revenue").Orientation = xlDataField

End Sub
```

The above macro will create an entire pivottable from data residing in the Jet3 external database. Let's now take a look at some of the properties and methods associated with the pivottable object

Pivottable Properties

Name

The Name property can be used to either set or retrieve the name of a pivottable. The following macro makes use of the PivotTable Collection to display the names of all pivottables on sheet1 in a message box:

```
Sub show_name()
   For Each pivot_table In Worksheets("sheet1").PivotTables
      MsgBox pivot_table.Name
   Next
End Sub
```

Range Properties

There are several properties of the PivotTable object that refer to specific ranges within the pivottable. These can be very useful for extracting data for various areas of the table. These range properties include:

RowRange - the range corresponding to the Row pivotfield
ColumnRange - the range corresponding to the Column pivotfield
PageRange - the range corresponding to the Page pivotfield
DataBodyRange - the range corresponding to the Data pivotfield

DataLabelRange - the range corresponding to the Labels of the Data PivotField
TableRange1 - the range containing the pivottable excluding the Page pivotfields
TableRange2 - the range containing the pivottable including the Page pivotfields

The macro below makes use of the TableRange2 property of the PivotTable object to delete an entire pivottable from a worksheet:

```
Sub Delete_Pivot()
   Worksheets("sheet1").PivotTables("pivot2").TableRange2.Clear
End Sub
```

RefreshDate
The RefreshDate property of the PivotTable object returns the date and time that the data of the particular pivottable was last refreshed. The line of code displays the date and time of the last refresh for pivottable Pivot2 in a message box:

```
MsgBox Worksheets("sheet1").PivotTables("pivot2").RefreshDate
```

Pivottable Methods

AddFields
The AddFields method is used to add fields to a pivottable. The AddFields method takes four parameters:

RowFields: database fields to be added to the row field of the pivottable
ColumnFields: databaes fields to be added to the column Field of the pivottable
PageFields: database fields to be added to the page field of the pivottable
AddtoTable: If True, new fields are added to existing fields. If False, fields replace existing ones.

The following line of code adds the database field "country" to pivottable Pivot2 in the row field - and the field is added in such a way that the existing fields remain in the pivottable:

```
Worksheets("sheet1").PivotTables("pivot2").AddFields RowFields:="country", AddToTable:=True
```

ShowPages
The ShowPages method creates a new pivottable for each item in the specified page field - each new pivottable is created on a new worksheet that is given the name of the item. The following line of code creates a new pivottable for each item in the Region page field in the pivottable Pivot2:

```
Worksheets("sheet1").PivotTables("pivot2").ShowPages "Region"
```

RefreshTable
The RefreshTable method refreshes the pivottable from the source data. The following example refreshes the data in pivottable pivot2:

Worksheets("sheet1").PivotTables("pivot2").RefreshTable

The PivotField Object

The PivotField object is a part of the PivotTable object and is used to represent a database field that is displayed in one of the four fields of the pivottable: row field, column field, page field and data field. The PivotField object provides the primary tool for manipulating data in the pivottable.

PivotField Properties

Orientation
The Orientation property is used to set the area within the pivottable where the particular pivotfield will be displayed. Orientation can take on one of five different values:

xlRowField: display pivotfield in the row field
xlColumnField: display pivotfield in the column field
xlPageField: display pivotfield in the page field
xlDataField: display pivotfield in the data field
xlHidden: hide the pivotfield from display

The following example displays the Region database field in the page field of the pivottable Pivot2:

```
Worksheets("sheet1").PivotTables("pivot2").PivotFields("region").Orientation = xlPageField
```

CurrentPage
The CurrentPage property is only applicable to pivotfields that are displayed in the page field of a pivottable. CurrentPage returns the item within the pivotfield that is currently being displayed in the table - this corresponds to the value that is displayed in the page field drop down list box. The following example shows the CurrentPage of the Region pivotfield in pivottable Pivot2 being set the the item "Africa":

```
Worksheets("sheet1").PivotTables("pivot2").PivotFields("region").CurrentPage = "Africa"
```

Position
The Position property is used to set the position of the pivotfield among all of the pivotfields that are in the same orientation - row, column, page or data. The following line of code changes the position of the pivotfield named "prod_cat" to the second position (for example, in the page field):

```
Worksheets("sheet1").PivotTable("pivot2").PivotFields("prod_cat").Position = 2
```

PivotField Methods

PivotItems

The PivotItems method returns a collection of all of the values within a particular pivofield - this collection corresponds to all the values in the database field that the pivotfield represents. The following code displays all of the pivotitems for the "product" pivotfield in a message box:

```
For Each x In Worksheets("sheet1").PivotTables("pivot2").PivotFields("product").PivotItems
    MsgBox x
Next
```

HiddenItems

The HiddenItems method returns a collection of items that are hidden from view in the pivottable. It is possible with the pivottable to specify that certain pieces of data within a pivotfield are not displayed - this is down by setting the Visible property of the PivotItem object to False. The HiddenItems method will return all of those PivotItems, within a certain pivotfield, for which the visible property has been set to false. The following line of code displays all of the hidden pivotitems in the "product" pivotfield:

```
For Each x In Worksheets("sheet1").PivotTables("pivot2").PivotFields("product").HiddenItems
    MsgBox x
Next
```

VisibleItems

In contrast to the HiddenItems method, the VisibleItems method returns a collection of all PivotItems that are visible within a pivotfield. The following example displays all such visible pivotitems for the "product" pivotfield in a message box:

```
For Each x In Worksheets("sheet1").PivotTables("pivot2").PivotFields("product").VisibleItems
    MsgBox x
Next
```

The Range Object

The Range object is used to reference cell ranges on the worksheet grid. It's primary use is in manipulating worksheet data - in applying complex formulas and calculations to large bodies of numbers. The Range object is also important in form design - in controlling placement of worksheet objects and controls as well as applying formatting and coloring to your forms. As with the other objects we have looked at, Range has several properties and methods - let's take a look at a few of them.

Locked

The Locked property can be used to lock a range of cells. Locking a range is only of consequence when a worksheet is protected - if a worksheet is protected, then all cells on the sheet that are locked cannot be edited or changed by the user. However, it would be possible to have a range of cells that are not locked that the user could freely edit even if the worksheet were protected. The following macro has the effect of unlocking a range of cells and then protecting the worksheet:

```
Sub protect_sheet()
   Worksheets("sheet1").Range("A1:D5").Locked = False
   Worksheets("sheet1").Protect
End Sub
```

Name

The name property is, perhaps, one of the most used properties of the Range obejct. In Excel, Range names are very useful in manipulating data on the worksheet - in moving data around, sorting data, calculating subtotals and performing calculations. The Name property is used to assign a text string to a Range as a name. The following example assigns a name to a range and then uses the range name to set the value of the range:

Range Methods

Calculate

The Calculate method forces a recalc on the specified range. This will force all Excel worksheet formulas to update all links in formulas:

```
Worksheets("sheet1").Range("a6:D10").Calculate
```

Clear
The Clear method has the effect of clearing the range:

Worksheets("sheet1").Range("a6:D10").Clear

Sort
The Sort method sorts data within the range. It is possible to pass sort keys to the Sort method to selectively choose which columns the data is to be sorted according to. The following example sorts data in the range A1:D10 based on a primary sort key in column A and a secondary sort key in column B:

Worksheets("sheet1").Range("A1:D10").Sort Key1:=Range("A1"), Key2:=Range("B1")

IV. Further Information

For further information, two books on Visual Basic for Applications are available through Microsoft Press:

1. **Excel VBA Step by Step** by Reed Jacobson (ISBN 1-55615-589-1, SRP $29.95) - self paced step by step lessons that take you from a basic introduction to VBA and the Excel Object Model to building real-world VBA applications.

2. **Excel 5 VBA Function Reference** (ISBN 1-55615-624-3, SRP $24.95) - this is a print-out (in book form) of all of the information included in the VBA Help file shipped with Excel 5.

These books can be ordered by calling 1-800-MSPress.