Accounting And Visual Basic: What's The Connection?

Mark G. Simkin, (Email: Simkin@equinox.unr.edu), University of Nevada
Nancy A. Bagranoff, (Email: Bagranna@muohio.edu), Miami University

Abstract

It is not enough today for accountants simply to know how to use word processing and spreadsheet software. In the knowledge age, accounting professionals must use information technology to the fullest. The ability to create, process, understand, and distribute information often determines work productivity, ratings on job performance evaluations, and even ultimate career successes. Accounting graduates with superior information technology (IT) skills are highly recruited, valued, and rewarded—these are the employees who are best able to perform the computer tasks required by their professional responsibilities. This paper reviews some reasons why today's accountants must be familiar with programming concepts in general, and Visual Basic (VB) in particular. It also reviews those VB programming tools that are especially useful to accounting applications.

Why Should Accountants Know Basic Programming Skills?

At one point in time, it was sufficient for accountants to have only rudimentary microcomputer skills—e.g., know how to use a limited word processor or create a simple spreadsheet. Those days are gone. Today, a familiarity with the latest version of an office suite is mandatory and, in fact accountants frequently need the full functionality in database and spreadsheet software to perform their work. The scope of technology skills has also expanded considerably, and now often includes: (1) a working knowledge of accounting or ERP and tax packages, (2) an understanding of the operating system software tools needed to find, create, copy, and delete files and file folders on different storage media, (3) the ability to use e-mail and groupware software, (4) expertise with the Internet as a research tool, (5) facility with presentation graphics software, (6) advanced knowledge of spreadsheet and database software, (7) knowledge of audit software tools, and (8) perhaps the ability to use specialized programs such as graphical documentation software [Bagranoff and Simkin, 2000].

Readers with comments or questions are encouraged to contact the authors via email.

In and of itself, the fact that accountants must know how to use personal productivity software does not automatically also require them to know how to program in a procedural programming language. Over the years, however, the functionality of personal productivity software—primarily word processing, spreadsheet, and database management software and perhaps to a lesser degree presentation graphics, operating system, email and groupware, and web browser software—has been enhanced with the availability of macro programming languages. These languages enable end users to create their own procedures and thus perform tasks that are unique to the application at hand. For the Microsoft Office Suite, this macro language is Visual Basic for Applications, or VBA, but other microcomputer software applications such as dBase, Lotus, and Novell have their own macro languages [Wildstrom, 1995].

Macro programming languages can help accountants in a variety of ways. A straightforward use is automating key-stroking tasks, thus freeing accountants from repetitious typing and enabling them to automate selected processing tasks. More advanced applications enable accountants to test input data for accuracy, spare novice spreadsheet users the need to understand complex filtering or table-lookup constructs, or audit spreadsheet models more accurately. Finally, macro programming language can act as a bridge between incompatible accounting software applications [Firester, 1994].

These capabilities also explain why macro programming languages also serve another important function—the ability to facilitate end user programming. Not long ago, accountants depended almost entirely upon the largesse of IT departments for custom computer work—for example, creating non-standard reports. With the availability of macro programming languages such as VBA, accountants can often download data from corporate data warehouses and/or file servers and create many of their own reports. But all this hinges upon a working knowledge of the programming tools required to perform these tasks—often Visual Basic.

Even if accountants leave most programming tasks to IT professionals, it often makes sense for them to understand the rudiments of computer programming. In the design of new computer systems, for example, one reason for this need is to avoid the high costs of poor system specifications. The idea that accountants can play an important role in designing the functionality of, and internal controls for, new AISs is well understood [Moscove, et. al., 1999]. This includes planning for new capabilities, interfacing with existing accounting systems, and of course, coordinating system implementation schedules so that they do not inconvenience employees during important accounting calendar events. What is less clear is the important role that accountants can play in creating the detailed design specifications for such systems, and the large costs that poor specs can cause. These costs include the wasted personnel costs caused by last-minute changes, delays in final systems implementation, and even lost jobs if things go badly enough [Lilly, 1998]. Similarly, it is almost impossible to test or audit a modern AIS without a detailed knowledge of what the system was designed to do. Again, precise design specifications provide exacting details for this, helping IT professionals, auditors, and end users find bugs before the system is completely installed and saving hours of unproductive time.

A working knowledge of computer programming is also useful to those accountants wishing to build their own Internet sites. Today, many CPAs realize that web sites not only help users locate external information or tax forms, but are also an important means of recruiting new employees, communicating with external clients, distributing expertise, and disseminating human resource information to employees. Knowledge of computer programming in general, and web site development in particular, can help accountants create such sites, or at least better design the specifications for one that an independent contractor creates [Cayton, 1966].

Finally, a side benefit to learning procedural-language programming is that it enhances the critical thinking and problem-solving skills that are vital to success in many accounting careers. Examples of such skills include deductive logic, systematic analysis and design, and forecasting. These skills are useful in a variety of non-programming accounting environments as well—for example, in performing forensic tasks or audit work.

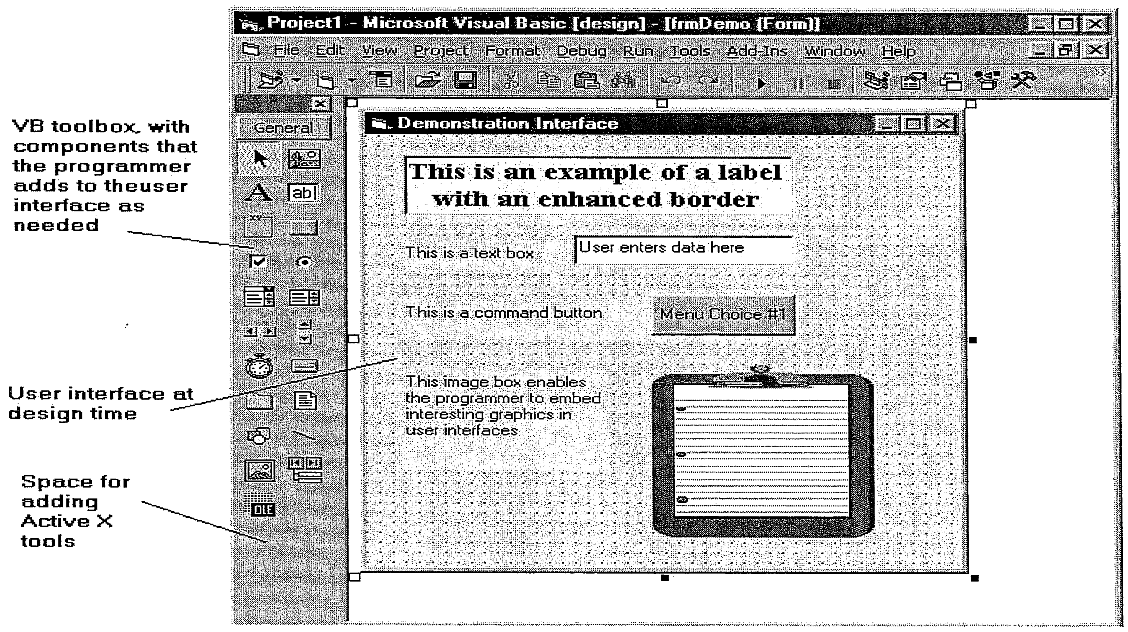
Visual Basic And VBA: What's The Difference?

Both Visual Basic (VB) and Visual Basic for Applications (VBA) are Windows-based, Microsoft products that allow their users to develop 32-bit, interactive, event-driven code. Here, the term "event-driven" means that what triggers a computer to execute code (i.e., program instructions) is usually a user event such as clicking on a Command Button, pressing the Enter key, or exiting a dialog box. Those accountants familiar with earlier versions of Basic language such as GW Basic or Quick Basic are likely to recognize fragments of modern VB code because they contain the familiar If tests, Do Loops, and assignment statements that were part and parcel of these predecessors.

It is important for accountants to realize that "Visual Basic" and "Visual Basic for Applications" are not quite the same. They are more like close cousins than fraternal twins. Visual Basic, for example, is a complete procedural computer language that allows programmers to develop familiar, Windows-based applications. The programming environment for VB is the VB Developer's Studio (Figure 1). This environment allows programmers to develop complete user interfaces in *design time*, and then test the application in *run time* to see if it operates correctly. Most Windows-based accounting software utilize the standard tools this environment provides—for example, the Labels (that provide headings, user instructions, or identifications), TextBoxes (for entering data), and CommandButtons (for triggering code) illustrated in the user interface portion of Figure 1.

In contrast, VBA is just that—a subset of Visual Basic tools that enable developers to create programming instructions for specific applications. VBA is accessible from all major applications in Microsoft Office 2000, but the programming environment is mostly the application itself—for example, a specific Excel spreadsheet—rather than a design studio [Grauer and Barber, 2000]. Even the names for the applications differ—"user interface" in the case of VB, "macro" in the case of VBA.

Figure 1 Visual Basic's Design Studio


VB toolbox, with components that the programmer adds to theuser interface as needed
User interface at design time
Space for adding Active X tools
Project1 - Microsoft Visual Basic [design] - [frmDemo (Form)]
File Edit View Project Format Debug Run Tools Add-Ins Window Help
General
Demonstration Interface
This is an example of a label with an enhanced border
This is a text box
User enters data here
This is a command button
Menu Choice #1
This image box enables the programmer to embed interesting graphics in user interfaces

The "applications orientation" of VBA code in Excel or Access is no accident. It allows end users to create their own applications without also requiring them to have the advanced programming skills of IT professionals. This orientation also allows users to focus on the task at hand—for example, testing the data in user entries—within the context of the worksheet or database table itself.

Perhaps the most important difference between VB and VBA to accountants is that VBA provides a *macro recorder*. This tool enables the system to "capture" user keystrokes, and generate Visual Basic code that represents the tasks or procedures these keystrokes perform. This enables the user to automate a specific process—for example, testing designated data entries for accuracy—and therefore also has the potential to ensure better data inputs and concomitant outputs. But it is as vital that an accountant understand the code such macro recorders generate as it is important that they understand how to use such recorders. This knowledge enables accountants to create and later modify their own macros, as well as attest to the accuracy of a client's macros, should this be required in an audit of the spreadsheet.

Understanding Visual Basic Code

Like other programming languages, Visual Basic programs are composed of three basic constructs: (1) *sequential or linear code*, with instructions that follow one another and that a computer executes in sequence, (2) *selection statements,* that enable a computer to choose between different processing alternatives, and (3) *looping or repetition statements*, that enable a computer to repeatedly execute a set of instructions until a specified exit condition is met. The discussions which follow examine each of these constructs in turn, focusing on Visual Basic coding examples that are typically found in familiar accounting environments and applications.

Linear Code

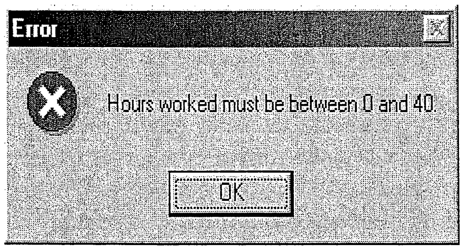
Linear instructions are programming statements that a computer executes in order. We shall look at four types of such instructions: (1) VB methods, (2) assignment statements, (3) computational instructions, and (4) instructions using functions.

VB Methods

Some of the simplest instructions, called *methods* in VB parlance, are those that perform simple tasks. Three examples are:

```
Beep        'Causes a computer to beep audibly
Print "This is what will be printed." 'Print
   something on the interface
MsgBox "Hours worked must be between 0
   and 40.",vbCritical, "Error"
```

The first instruction causes a computer to make a sound—the familiar "beep" that users hear when they make an error. This beep is an important internal control over data entry, referred to as a program edit check. Accountants can control data input, for example, in a *Microsoft Access* database application by limiting the acceptable data input set in a form. The database developer can program the Beep in VB or the software will do it automatically based on the specified validation rules in the database model. The second instruction prints the words in quotation marks directly on the user interface. The third instruction causes the computer to display the MessageBox shown in Figure 2. (A MessageBox uses three parameters, the first of which is the message itself, the second of which determines what icons and CommandButtons appear in the window, and the third of which determines the heading at the very top of the window.) The text to the right of these instructions beginning with apostrophes, are *comments* that the com-


Error

Hours worked must be between 0 and 40.

Figure 2 An example of a MessageBox

puter will ignore during execution, but which are nonetheless useful for documenting the code and explaining what is accomplished. Such comments make programs easier to fix or modify later.

Assignment Statements

A second example of sequential code is an assignment statement that sets a processing variable, TextBox, or other Visual Basic control to a specified value. An example of such an instruction that might be recognizable to those familiar with VBA coding those is:

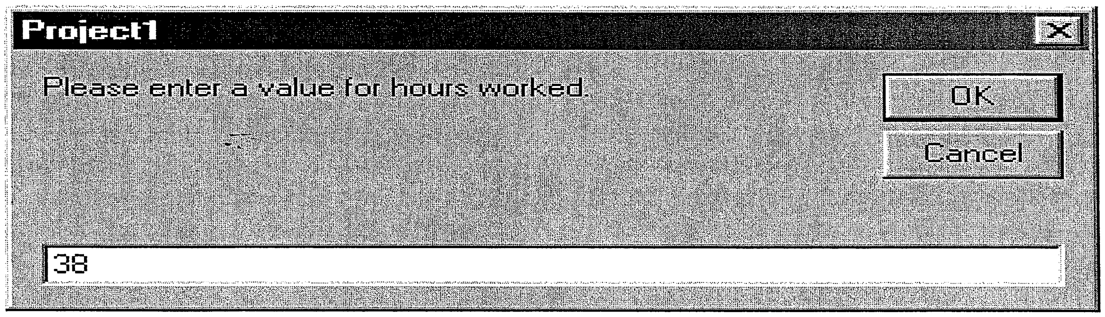
nHoursWorked = InputBox("Please enter a value for hours worked.")

This instruction causes a computer to display the simple dialog box illustrated in Figure 3—an InputBox. (The difference between an InputBox and a MessageBox is that an InputBox allows the user to input a value, whereas a MessageBox merely displays a message—the user cannot enter new data with it.) As you may already know, both InputBoxes and MessageBoxes are *modal*, meaning that the user cannot continue until he or she has responded to the dialogue box—for example, by clicking on the OK button.

A second type of assignment statement is one that enables a programmer to assign values to programming variables—i.e., to elements that the end user cannot see, but which nonetheless enable a program to function. These are good internal controls and examples are:

```
nHoursWorked = 40        'Set hours worked
  to a default value of 40
PayRate.Text = 9.75      'Set pay rate text-
  box to $9.75
nPayRate = nMinRate      'Set the pay rate
  variable to the specified minimum
```

Assignment statements initialize the variable or other element on the *left* side of the equation to the value on the *right* side of the statement. Thus, in the first example, the instruction initializes the variable nHoursWorked to "40."


Project1

Please enter a value for hours worked.

38

Figure 3 An example of an InputBox

You might wonder about the difference between the second or third instructions in this example. The second example sets the value of a TextBox (which the user can see in a user interface and perhaps modify) to "9.75." In contrast, the third example sets a variable called "nPayRate" (which the user cannot see) to the value of a second variable called nMinRate. Why set one variable equal to another? In this case, nMinRate is a *parameter value* that the program initializes once at the beginning of the program, and then uses repeatedly throughout the application as needed. If the nMinRate changes (for example, due to a change in state or federal law), the modifications required to reflect this change are nominal—the new value need only be entered once. This again helps make it easier to modify computer programs as needed.

Computations

Yet another type of linear code is a statement that performs a computation. Two examples are:

```
nGrossPay = nHoursWorked * nPayRage
          'Gross pay if hours worked < 40
nGrossPay = 40*nPayRate + 1.5*(nHoursWorked - 40)* nPayRate
  'Gross Pay if overtime
```

The first instruction computes the gross pay if an employee works 40 hours or less, and the second instruction computes the gross pay if an employee works more than 40 hours *and* earns time and a half for overtime.

The functionality and processing capabilities of most accounting programs are usually determined by assignment statements such as these that perform calculations. Surprisingly, however, the vast majority of instructions within VB programs are *not* computational. Rather, they do other things—especially test data for accuracy and completeness, two data characteristics with which accountants are particularly concerned.

Using Functions

There are many other examples of linear statements, but we shall only look at one final example—statements that use functions to overcome *data type problems*. To begin, it is necessary to know that Visual Basic enables its users to store numerical data in a large number of different formats called *data types*. Examples include Integer, Single Precision, Double Precision, Long Integer, String (Text), and Currency. The selection of a specific data type tells Visual Basic how to store the data—i.e., the specific bit configuration to use for the data element.

Although the intricacies of such data stores are probably of little interest to accountants, they are nonetheless important because the choice of data type sometimes determines the way in which Visual Basic performs computations as well as almost always determines the format in which Visual Basic outputs computed results. To illustrate, suppose you input two numerical values in each of two InputBoxes and then attempt to add the two values together. The code looks like this:

```
nRegularHours = InputBox ("Input the Number of Regular hours worked.")
nOverTimeHours = InputBox("Input the number of overtime hours worked.")
nTotalHours = nRegularHours + nOverTimeHours
```

If the user inputs "40" for regular hours and "5" for overtime hours, the unexpected total is 405! The reason is because InputBox values are *string* data types, not *numerical values*. Thus, when Visual Basic encounters the third instruction, it will *concatenate* the strings rather than *add* the values together. To overcome this problem, the programmer must use Visual Basic's Val function to perform the math as follows:

```
nTotalHours = Val(nRegularHours) + Val(nOverTimeHours)
```

Here, the Val function will first convert nRegularHours to a numeric value, which Visual Basic will then interpret correctly and manipulate accordingly.

Developers will encounter a similar data type problem when they use certain Visual Basic data types to prepare outputs. For example, consider the following VB instruction:

```
txtOutput.Text = 40/3
```

This instruction divides 40 by 3, and then displays the results in a TextBox. But the resultant output is "13.33333," even if the user wants only two-decimal place accuracy.

To overcome such problems (as well as several others), Visual Basic provides a Format function (in both VB and VBA) that enables users to convert numerical values from one data type to another. An example is:

```
TxtOutput.Text = Format(40/3, "Currency")
```

In this example, the value to display is the first argument in the function (i.e., "40/3") and the second argument is the data type setting that Visual Basic should use for this (i.e., "currency"). The resultant output is "$13.33." Other formatting settings include Integer, String, or Date. Visual Basic also supports the custom data masks familiar to Access users. An example is:

```
txtOutput.Text = Format(40/2, "###.00")
```

In this example, the mask "###.00" uses pound signs to the left of the decimal point as place holders, and the zeros to the right of the decimal point as values that Visual Basic will replace with significant digits if it computes any, or leave as "0's" otherwise. This explains why, for the current example, the resultant output will be "20.00"—not "20."

Those accountants who dismiss VB and VBA formatting capabilities as esoteric trivia or of little value are in for a big surprise when they attempt to create mailing lists using *Access* and *Word*. In order to export numeric values (such as averages) that are computed from the data fields of an underlying database to Microsoft Word in a desired format, for example, the *Access* query must first contain exactly such a Format instruction in order to display the calculations properly in the letter.

Selection Code

Selection instructions enable computers to make decision choices—for example, to determine which of several processing alternatives to use. A key example in accounting applications is to test input data for accuracy and completeness. As a general rule, the data in accounting databases must be completely, numbingly accurate because the costs of inaccurate data (e.g., an incorrect credit card number) are so high. Thus, the processing task is to examine a specific input value and decide whether or not to accept it.

The two most common selection statements in Visual Basic are the *If tests* familiar to many Excel and Access users, and *Select Case statements*. In accounting applications, one of the most common uses of selection statements is to determine whether or not an input value is acceptable. An example of an If clause, which includes both the If test and the processing to perform if the test proves positive, is:

```
If txtHoursWorked.Text = Clear Then    'Test
      for blank value
   Beep              'provide audio feedback
   MsgBox  "Error.  Please enter a value for
      hours worked."
   Exit Sub                  'Exit subroutine
End If
```

The *If test* in this example is testing for a blank TextBox. (The word "Clear" is a VB reserved word that tests for null values.) If this condition is true, the computer will execute all the other (indented) instructions within the clause. If this condition is false, the computer will ignore the entire clause and proceed to the next instruction following the "End If" statement. As you can see, the instructions in this example test for a blank TextBox, and indicate what to do if the user provides one. Other examples of data validation tests include range tests, matching tests, tests of data completeness, and tests for valid codes—see Figure 4.

Where there are several acceptable data entries for a given TextBox or InputBox, programmers often find it easier to use a Select Case clause to test for them. For example, suppose a computer application must test a state code, and compute the sales tax according to the user's home state. A (simplified and fictitious) example of such a situation would be:

```
Select Case txtStateCode.Text          'state
     code is the selector variable
  Case "NY", "NJ", "VT", "WV", "HI"
    nTaxRate = .08
  Case "MS", "OK", "NV", "CA", "FL",
     "GA"
    ntaxRate = .085
  Case "MA", "CT", "NC", "CO", "UT",
     "AZ"
    nTaxRate = .09
  Case Else
    Beep
    MsgBox "Error. Unrecognizable state
      code. Please reenter."
    Exit Sub
End Select
nTaxAmount.Text = nTaxRate * nPur-
     chaseAmount
```

Figure 4 Examples Of Data Entry Validation Tests

Type of test	Example of If Test*	Comments
Test for Blanks	If txtHoursWorked = Clear then	"Clear" is a Visual Basic key word
Range Test	If nHours <0 or nHours > 40 then	Tests for hours less than 0 or more than 40
Matching Test	If nPass1.Text < >nPass2.Text Then	Tests whether a user enters the same password correctly in two successive TextBoxes.
Completeness Test	If Len(nSSN) < > 9 Then	Len function computes the length, in bytes, of a variable. This example tests for a nine-digit social security number
Code Test	If nGender < > "M" And nGender < > "F" Then	Gender must be either "M" or "F"

* Note: "< >" means "not equal to" in Visual Basic

In this example, Visual Basic uses the state code in the TextBox called "txtStateCode" as the selector variable—i.e., the variable to test. If this value matches any of the state codes in the first Case (i.e., NY, NJ, VT, WV, or HI), Visual Basic will set the tax rate variable (nTaxRate) equal to 8 percent and then exit the Select clause. Similar logic applies to the second and third Case possibilities. Finally, if the system does not find any matches, the "Case Else" portion of the Select Case clause triggers. Here, we see our now-familiar error routine instructions, which cause the computer to Beep, display an error message, and exit the procedure.

Loop Code

As noted above, a third possible programming construct creates processing loops—i.e., instructions that execute repeatedly until the computer encounters a termination condition. These looping constructs, for example, enable a computer to prepare payroll checks for a set of employee time cards, or compute the present value of a stream of future payments.

Two important VB loop clauses are For-Next Loops and Do-While Loops. To illustrate, let us assume that an accounting application must compute the present value of a stream of equal future values. The formula is:

$$\text{Present Value} = \sum_{k=1}^{n} A/(1+R)^k$$

where n = the number of years, A = the amount received each year, and R = the interest rate. Although Visual Basic provides a function for this, it is also instructive to see how one might program this in Visual Basic using a For-Next loop. The following instructions assume that the variables nYears, nRate, and nAmount have been initialized elsewhere in the program:

```
nSum = 0
For K = 1 to nYears
    nSum = nSum + nAmount/(1+nRate)^K
Next K
txtPresentValue.Text = Format(nSum, "Currency")
```

In this code, the For and Next instructions create a repetitive loop, which Visual basic will execute the prescribed number of times—for example, five times if nYears equals "5." In each successive pass through the loop, the system will compute a value for "nAmount/(1+nRate)^K" (the ^ stands for exponentiation), and successively add this value to the accumulator variable nSum. Finally, after completing its work, the system will exit the loop and execute the instruction immediately below the Next instruction. That final instruction will display the results in the TextBox called "txtPresentValue," format in currency fashion as described above.

A Do-While Loop is similar to a For-Next loop in that it, too, creates a set of instructions that a computer will execute repeatedly until it reaches a termination condition. A simplified example of such a loop is:

```
Do While Not EOF(1)
  [other processing instructions]
Loop
```

In this example, the termination condition is "Not EOF(1)"—i.e., "not end of file #1." There will typically be a large number of processing instructions inside such a loop, but the very final one will always be "Loop"—which tells Visual Basic to transfer control to the very first instruction and test for end of file again. If it has not been reached, the system will execute the internal processing instructions yet again, and repeat this process until end of file *is* reached.

Summary And Conclusions

There are many reasons why accountants should have basic programming skills and be familiar with programming concepts. High among them is to be more computer literate, functional, and productive when using the macros that accompany such office productivity tools as Microsoft Word, Excel, or Access. Other reasons include the need to help clients with end-user programming tasks, audit the macro instructions found in client spreadsheets or databases, better prepare the controls and system specifications for larger computer systems, create better, more functional web sites, and improve their critical-thinking and problem-solving skills.

Visual Basic (VB) and Visual Basic for Applications (VBA) are not the same, but the differences are nominal. Visual Basic is a complete, procedural programming language that IT professions typically use to create fully functional systems for end users. Visual Basic for Applications is a subset of VB, and is typically employed by end users to accomplish processing tasks for themselves.

The final portion of this paper reviewed three types of Visual Basic instructions: (1) linear code such as VB methods and assignments statements that use computations or functions, (2) selection code such as If tests and Select Case statements, and (3) looping code such as For-Next statements and Do-While statements. Understanding these constructs can help accountants create better macros for themselves, better audit the macros of their clients, and better understand how computers test input data for accuracy and completeness.

Bibliography

1. Bagranoff, Nancy A. and Mark G. Simkin "Picture That: How to Tell the Story With Graphics" *Journal of Accountancy* (February, 2000), pp. 43-46.
2. Cayton, Brian "Build Your Own Internet Site" *Accounting Technology* Vol. 12, No. 9 (October, 1996), p. 41ff.
3. Firester, Jonathan "Accounting Software Macro—Painting Yourself Out of a Corner" *Accounting Technology* Vol. 10, No. 11 (December, 1994), p. 8.
4. Grauer, Robert T. and Maryann Barber "A VBA Primer: Extending the Power of Microsoft Office" (Upper Saddle River, New Jersey: Prentice Hall, 2000).
5. Lilley, Vic "How to Avoid the High Cost of Poor Specifications" *Accounting* (December, 1998), p. 26.
6. Moscove, Stephen A., Mark G. Simkin, and Nancy A. Bagranoff *Accounting Information Systems* 6th ed. (New York: John Wiley and Sons, 1999).
7. Wildstrom, Stephen H. "Programming Without Tears" *Business Week* Issue 3418 (April 3, 1995), p. 18.