Bitter, Rick et al "ActiveX"

*LabVIEW Advanced Programming Techniques*

Boca Raton: CRC Press LLC, 2001
As many readers have known and suspected, there is a lot more to ActiveX than a LabVIEW interface. ActiveX is a relatively new technology that wraps around several other technologies, namely OLE and COM. This chapter gives a general outline of ActiveX, COM, and OLE technologies and how they may be applied by the LabVIEW programmer using versions 5.0 and 5.1.

This chapter covers only the Windows 95, 98, NT, and 2000 versions of LabVIEW; Windows 3.1 does not support COM, ActiveX, or OLE Version 2. The core of ActiveX, COM, is currently being ported to other operating systems. As the COM subsystem is ported to other operating systems, support for ActiveX will follow. The political nature of competing operating systems’ vendors suggest that one shouldn’t hold one’s breath, however.

Component Object Model (COM) is a new programming paradigm for software development. Microsoft is providing a path for developers to practice component-based development. The mechanical and electrical engineering communities have long practiced this. When designing a circuit board, resistors (components) are chosen from a catalog and purchased for use on the circuit board. Software developers have never truly had a component-based development environment.

Libraries, both static and dynamically linked, have been available for quite some time, but were never well suited for component development and sale. Many companies are unwilling to distribute information about code modules such as header files or the source code itself. Dynamically-linked libraries have proven difficult to work with since no mechanisms for version control are provided. Libraries may be upgraded and older applications may not be able to communicate with the upgraded versions. The ActiveX core, COM, is a standard that attempts to address all the major issues that have prevented software component development and sale. The result is expected to be a large pool of components that programmers may assemble to create new applications. Ultimately, this chapter will provide the background needed for Win32 LabVIEW developers to take advantage of the new component architecture.

LabVIEW itself may be called as an ActiveX component, and may call other components. The VI Server has two implementations, one as an ActiveX control for Win32 usage, and a TCP-based server that may be used by LabVIEW applications on non-Windows platforms for remote control. The VI Server feature will also be covered in this chapter.
7.1 INTRODUCTION TO OLE, COM, AND ACTIVEX

In the beginning, there was Windows 3.1 and the clipboard. The clipboard was the only means for Windows applications to exchange data. Windows 3.1 then supported DDE and OLE Version 1. Dynamic Data Exchange, DDE, was a means for applications to communicate and control each other. OLE represented Object Linking and Embedding, a mechanism for applications to present data belonging to other applications to users. OLE also provided support for drag-and-drop files. These were fairly significant advancements.

Visual Basic also introduced the .vbx file, Visual Basic eXtension. A Visual Basic Extension was generally a control, similar to the controls we use in LabVIEW. These controls helped expand the range of applications that could be written in Visual Basic. The functionality of the VBXs proved to be wildly popular. VBXs worked well in the 16-bit Windows 3.1 world, but were only usable by Visual Basic. When 32-bit Windows 95 and NT 4.0 came out, it turned out to be difficult to port the 16-bit VBX controls. The decision to redesign the VBX control was made, and it was determined that OLE was the mechanism that would support it. Beneath OLE was COM, the component object model.

COM, OLE, and ActiveX have become somewhat stable these days. Programming in Windows with new technologies can be troublesome, as Microsoft tends to evolve technologies after they have been released. ActiveX developers know and understand that the COM standard is well defined and has not changed much, but the implementation of this technology has changed quite a bit in the last few years. Support for ActiveX and COM development now exists in the Microsoft Foundation Classes (MFC) and the ActiveX Template Library (ATL). The ATL has evolved in the last couple of years, and developers also understand that code development between Visual C++ 5.0 and 6.0 has changed. As ActiveX users, we do not need to be concerned with the implementation issues; all we care about is that properties and methods can be invoked regardless of what the developers need to do to make it work. If you choose to develop your own ActiveX controls, you will need to do some research on which development tools and libraries to use, such as Visual Basic or C++. Additionally, be aware that COM+ has been released with Windows 2000. There are potential issues for developers with this new release of COM.

7.1.1 DEFINITION OF RELATED TERMS

This section will introduce a few definitions that are commonly used when working with COM technologies. Several terms are introduced in Chapter 9, Multithreading, and will be used again in Chapter 10, Object-Oriented Programming.

7.1.1.1 Properties and methods

All ActiveX controls have some combination of Properties and Methods that are available for other applications to control. Properties are variables that are exposed to outside applications. Properties can be configured to be readable only, writable only, and read-write. The act of reading a property is referred to as a GET; writing
a variable is a \textit{SET}. The ActiveX control maintains a list of properties that may be accessed by external applications.

Methods are functions that are executable to external applications. Methods may require arguments and have return values. Not all functions internal to an ActiveX control may be called by external code. The ActiveX control maintains a list of functions it exports. Calling a method is referred to as “invoking” the method.

7.1.1.2 Interfaces

An interface is a group of properties and methods. ActiveX controls may support (and always do) more than one interface. One such interface that all COM objects support is IUnknown. This interface is used to identify the location of other interfaces, properties, and methods. The rules of COM require that once an interface is published, it cannot be changed. If programmers want to add functionality to a COM object, they must add a new interface and make changes there.

All COM objects have one interface that is identified as the default interface. The interface LabVIEW uses is selected when the user inserts the control into LabVIEW. Some controls have a single interface; other controls, such as for Microsoft Access, support hundreds of functions spread across dozens of possible interfaces.

7.1.1.3 Clients and Servers

COM and its derivative technologies, OLE and ActiveX, use a client and server connection to communicate with LabVIEW. When LabVIEW is accessing an ActiveX control, LabVIEW is the client application and the ActiveX control is the server. As a client, LabVIEW becomes capable of executing methods and accessing properties of the ActiveX controls that LabVIEW “owns.”

When LabVIEW is being controlled via an external application, LabVIEW becomes the server. Other applications, which may or may not be running on the same machine, use LabVIEW’s services. Other clients can include Visual Basic applications, applications written in C++, and even Microsoft Office documents with scripting support that can control LabVIEW.

7.1.1.4 In-Process and Out-of-Process

An In-Process ActiveX control is equivalent to a DLL that is loaded into the same memory space as LabVIEW. Chapter 9, Multithreading, discusses protected memory. When an ActiveX control is loaded into the same memory space as LabVIEW, it falls under the complete control of LabVIEW. When LabVIEW exits, the ActiveX control must also exit, and the memory space it resides in will be reclaimed by the system.

Out-of-Process controls are independent applications that are controlled through the COM subsystem. A special DLL called a “proxy” is loaded into LabVIEW’s memory space. Calls to the external object are made through this proxy. The purpose of the proxy is to make it transparent to programmers whether the control that is being programmed is in-process or out-of-process. Passing data between processes is a much more difficult task than many people appreciate, especially when data
with unknown size (strings) are being transported across process boundaries. The proxy uses a mechanism called “marshalling” to pass data between processes. For more information on processes and protected memory space, refer to Chapter 9.

7.2 COM

The core of the Component Object Model is a binary interface specification. This specification is language- and operating system-independent. Attempts are currently underway to port the COM specification to other operating systems such as UNIX. COM is the foundation of a new programming paradigm in the Win32 environment. ActiveX and OLE2 are currently based on the COM specification. Neither technology replaces or supercedes COM. In fact, both OLE and ActiveX are supersets of COM. Each technology addresses specific programming issues, which will be discussed shortly.

Because COM is intended to be platform- and machine-independent, data types are different in COM than standard types in the C/C++ language. Standard C/C++ data types are anything but standard. Different machines and operating systems define the standard types differently. A long integer has a different interpretation on a 32-bit microprocessor than on a 64-bit microprocessor. Types such as char are not defined in COM interfaces; a short, unsigned data type is defined instead. COM defines its types strictly and independently from hardware.

COM does define a couple of data types in addition to primitive types used by nearly every language. Date and currency definitions are provided as part of the interface. Suprisingly, color information is also defined in COM. OLE and ActiveX controls have the ability to have their background colors set to that of their container. The information identifying background colors is defined in the COM specification. COM also uses a special complex data type, the variant.

7.2.1 The Variant

Programmers with experience in Visual Basic, Visual Basic for Applications, and Visual Basic Script have heard of a data type called the variant. Variants do not have a defined type; these variables are meant to support every data type. In high-level languages such as VBScript, the variant allows for simple, fast, and sloppy programming. Programmers are never required to identify a variable type. The variant will track the type of data stored within it. If a variant is reassigned a value of a different type, it will internally polymorph to the new type. ActiveX, COM, and OLE controls allow the usage of variants, and therefore they must be supported in LabVIEW. We’ve encountered a number of issues with the variant type in LabVIEW, and this section will help explain what this data type is and how it works “under the hood.”

From a low-level language such as C++, variant programming can be difficult. The variant type has a number of binary flags that are used to indicate what type of data it contains. For example, variants can contain integers or arrays of integers. A C++ programmer needs to check binary flags to determine which primitive type (integers) and if any flags such as array are set. LabVIEW programmers go through similar problems; we need to know what type of data the variant is supposed to
contain and convert it to G data. The “To G Data” function is explained later in this chapter. LabVIEW programmers do not need to check for unions and bit-flags to work with Variants. We do need to know what type of data an ActiveX control is going to pass back, however. If you choose the wrong type, LabVIEW will convert it and give you something back you probably do not want.

Variants do not support every type of data. For example, it is not possible to represent defined pointers, C++ class objects, or structures in a variant. Advanced programmers will understand that void pointers can be contained in variants. Void pointers are pointers to memory addresses that do not contain information as to what is being pointed at. The programmer must know what a void pointer references before they can use it. The reason pointer operations are not supported in COM is because not every language supports the use of pointers. For example, LabVIEW programmers never need to concern themselves with memory addresses. To support COM's design to be platform- and language-independent, variants support primitive data types and a number of types defined in the COM standard. For example, dates and currencies are defined data types that COM supports.

Strong words of warning need to be communicated if you are dealing with variant strings. The problem that needs to be understood by the LabVIEW programmer relates to Unicode versus ANSI strings. The standard string type is ANSI, a string being represented by an array of 8-bit unsigned integers. A new standard, Unicode, was introduced because eight-bit strings are not very useful for languages other than English. Unicode strings consist of 16-bit unsigned integers. When converting strings to LabVIEW strings, you may have the unpleasant surprise of finding that the variant flattened out into an empty string. This is because LabVIEW was trying to convert a 16-bit array into an 8-bit array. This will not work in LabVIEW, and it took us several days of examining code to identify the nature of the problem.

Windows 95 and 98 are built on handling ANSI strings (eight bits), with one exception. Windows NT is built entirely on Unicode strings (16-bit). Windows NT will do Unicode-to-ANSI string conversions internally when necessary. The string translation will be transparent to the programmer and user. The one exception to ANSI strings in Windows 95 and 98 is the OLE2 library. COM, ActiveX, and OLE use this library, which is built entirely on Unicode string handling.

When using variant strings in LabVIEW, they MUST be converted to arrays of eight-bit integers and then converted to strings. If this is not done, you will always receive empty strings from a variant conversion. LabVIEW will otherwise try to convert the 16-bit string as an 8-bit string. When this is attempted, the string will always return empty. The variant’s first eight-bit value will be hex 0; the zero will be interpreted as a NULL string, a string with zero length. Converting the variant to an 8-bit array will force the conversion of the 16-bit numbers to 8-bit numbers, and the string will be recovered.

Communications applications rarely use 16-bit characters. UDP, TCP, Serial, and GPIB data is always an eight-bit character base. Microsoft built Windows NT and the OLE2 library on Unicode strings to address another issue that is important in the world today. ASCII strings work well when your primary language is English. If your primary language is not English, but is based on the same alphabet as English, such as Spanish, then ASCII is not that bad. If you happen to be from the Far East,
then ASCII strings can be quite difficult. You are required to either wait for custom versions of Windows for your native language to become available or use English versions. Unicode addresses this, by providing a base character that can cover over 65,000 possible letters. This is not a perfect solution, but it does address computer issues that impact a majority of the world’s population.

7.2.2 Problems that COM Addresses

COM addresses a number of component development issues. One such issue is interface requirements. The COM specification demands that once an interface has been shipped, the interface is “cast in stone.” If additional functionality is desired, it must be added in an additional interface. COM does allow an object to present multiple interfaces. This has been a characteristic problem with DLL development. When a DLL interface is changed, many older applications will not be able to cope with the new interface and will crash. All COM objects provide IUnknown, a specific interface. This allows a COM-based application to have a guaranteed entrance point to the object to identify what functionality the object provides. LabVIEW programmers who develop or use custom DLLs can implement the DLL functionality as ActiveX/COM objects and use the code in multiple applications. Environments such as Visual Basic have shown that ActiveX component programming is fairly easy, easier than using DLLs. The interface defined by COM and expanded on by ActiveX is intended to simplify programming.

Since the COM specification defines a common interface, why not expand the model to distributed computing? Microsoft did, and the result is DCOM—Distributed COM. This is a concept that DLLs alone could never have provided. Without the common DCOM specification, developers would have to develop custom interfaces for distributed application development. The use of a distributed object model offers many potential benefits to LabVIEW developers who program in an environment utilizing multiple computers.

7.2.3 In-Process and Out-of-Process COM Objects

COM objects may be represented in one of two manners: DLLs and executable applications. A DLL is linked into an application when it is started up. A COM object implemented as a DLL is referred to as an in-process COM object. In-process COM objects are loaded into the same memory space as the client application. A COM object implemented as an executable application is an out-of-process COM object. In order for the client application to access the COM object’s methods, an interprocess communication must occur. In general, communication with in-process COM objects is faster than out-of-process COM objects. When data crosses a process boundary, the operating system must become involved. As processes are switched into and out of physical memory, different threads of execution take over the CPU. This procedure generates additional overhead. This does not happen with in-process COM objects. The fact that out-of-process communication occurs should not dissuade a LabVIEW programmer from using COM servers. This is something that developers will need to be aware of when developing time-critical applications.
One advantage out-of-process servers have is stability. When an application is started, it is given its own address space and threads of execution. In the event that one process crashes, the other processes will continue to execute. These other processes may experience problems because a peer crashed, but good exception handling will keep other processes alive to accomplish their tasks. It is possible to crash peer processes, but defensive programming can always mitigate that risk. Out-of-process servers can also be run on different machines using DCOM, which allows for the process to continue executing even if one machine crashes. There are applications in monitoring and production software for distributed computing.

### 7.2.4 COM Object Identification

COM Objects have two forms of names: a text-based name that programmers and users can identify, and a Globally Unique ID (GUID). The GUID is a 128-bit number that is unique. The compiler assigns the GUID when the control is initially generated. Among other pieces of information that are used are the time and machine address of the computer that generated the control. This is not a completely foolproof scheme, but to a very high probability, the GUID number will be unique in the world. The GUID is stored in the user computer’s registry and is used by the COM subsystem to identify the object and its location on a hard drive or on the Internet. Programmers rarely, if ever, use the GUID, and this will not be discussed further.

COM Objects have an object-type library file associated with them. This library file identifies methods, properties, and interfaces the object supports. COM objects must also be registered with the system. The Win32’s registry will contain an entry identifying the path to the object and type library. When an application requests a connection to the object, it contacts the operating system, and the operating system provides the linkage. The application will read the type library, which identifies the methods that the object supports. The type library uses a specified language called IDL, Interface Description Language. IDL descriptions are compiled to become the type library. The use of COM objects and the type library is more stable than the standard use of DLLs. When a bad link is made to a DLL, the application will probably crash. If your application contacts the operating system to link to a COM object, and something is not working, the operating system will respond with an error message. The COM object itself may be in a different process or machine, and can also provide extra levels of isolation.

When a programmer identifies an ActiveX class while programming, the application shows a list of interfaces the control supports. This information is read from the type library. In reality, the operation is much more complicated and beyond the scope of this discussion.

Readers familiar with C and C++ recall the use of pointers. COM uses a virtual table that is exposed by the IUnknown interface. This table is not built of the functions the COM object supports, but is built of pointers to the functions. This double indirection actually gives COM an advantage. One of the rules of the COM specification requires that interfaces never be changed once they are published. Each and every ActiveX object you use in your LabVIEW applications will not need to be changed, ever. Regardless of what developers do to the objects you use, the
interfaces will not change. COM does give the developers flexibility as to what happens internally to the COM object. The virtual table can point to different functions. The table itself will change structure. Again, standard DLLs do not have this requirement; you may have to make changes to each application that uses a DLL that has been changed.

7.2.5 HOW COM OBJECTS ARE CALLED AND USED

All COM objects present an interface called IUnknown. This interface is used to describe the COM object itself, and performs a multitude of functions. Most of the IUnknown interface is well beyond the scope of this book. In fact, the average ActiveX programmer does not write their own IUnknown interface; they use code generated by Microsoft tools. For example, when writing a control in Visual Basic, the programmer never sees code related to the IUnknown interface. COM also has several other interfaces, but as LabVIEW programmers we do not need to be concerned with these other interfaces’ existence.

COM objects export their functions in a type library. The type library defines what properties and methods are available. Each property defines if it can be read or written, and functions indicate their argument lists and return types. The type library is compiled and cannot be read by a programmer. Type libraries are written in either Object Description Language (ODL) or Interface Description Language (IDL). IDL is the more recent language and is generated by compilers for COM objects. The IDL files are compiled into the type libraries.

This section will discuss two methods that applications use when linking to ActiveX controls: early and late binding. Binding is a process by which sections of code integrate themselves. There is a lot of behind-the-scenes pointer handling, and the process can get quite messy. Early binding means that the application and ActiveX control are linked together when the application is compiled. This is a fairly fast startup, functionally the ActiveX control (an .ocx file) is a DLL that is being merged at startup. The compiler uses the type library to build the application.

Late binding is the process by which the ActiveX control is loaded in while the application is executing. As mentioned in the last paragraph, an OCX file is a DLL with some specific interface functions. In general, this is a slower process than early binding. Late binding is an important concept for LabVIEW programmers, because this is how LabVIEW binds to ActiveX controls. When an application performs late binding, the operating system is asked to query the registry about the component. Late binding uses a different interface, and a Virtual Table is set up. A “vtable” is simply a list of pointers that are used to identify functions contained inside the control.

Conceptually, this is an extremely messy operation, and it is generally covered up by the programming language’s code. For example, LabVIEW and Visual Basic programmers never need to know or worry about the pointer tables being set up. Surprisingly, C++ programmers have little to worry about also. The MFC classes
provide a lot of this functionality. The C++ programmer has the freedom to replace the supplied interface code, however.

Late binding is not significantly slower than early binding. What can hurt the programmer is the registry query and loading the control into memory. Disk access can be slow when compared to the execution speed of the application. The actual overhead of a late-bound control is going through the virtual table to identify the starting memory address of a function. With modern CPUs running at speeds over 300 MHz, a few extra clock cycles is negligible in a vast majority of applications. Real-time programmers are the only ones who should be considering this. This is important to the LabVIEW programmer because all ActiveX controls you use will be bound late. When writing applications using ActiveX, always start up the controls at the start of your VI. This will eliminate the chance that an ActiveX control will slow down the application at sensitive times.

7.3 OLE

We will demonstrate in Section 7.4 that OLE interfaces are in fact, ActiveX objects. The reverse is not always true; some ActiveX objects are not OLE interfaces. OLE is an interfacing technology used for automation, the control of one application by another. Many programmers subscribe to the myth that ActiveX has replaced OLE—this is not true. OLE is a subset or specialization of ActiveX components. In general, ActiveX will be more widely used. LabVIEW applications that are controlling applications like Microsoft Word and Excel are using OLE, not ActiveX.

A document is loosely defined as a collection of data. One or more applications understand how to manipulate the set of data contained in a document. For example, Microsoft Word understands how to manipulate data contained in *.doc files and Microsoft Excel understands .xls files. There is no legitimate reason for the Word development team to design in Excel functionality so that Word users can manipulate graphs in their word documents. If they did, the executable file for Word would be significantly larger than it currently is. OLE automation is used to access documents that can be interpreted by other applications.

7.3.1 Origins and Applications

OLE once meant “Object Linking and Embedding,” but this is no longer the case. OLE was initially designed to allow embedded documents, and still serves this purpose. To an extent, OLE was designed to replace Dynamic Data Exchange. LabVIEW 5.0 still supports DDE, but it no longer has an OLE palette. The OLE palette is no longer necessary because OLE functions operate exactly like ActiveX functions. While DDE is still available to the LabVIEW programmer, its use should not be designed into new applications. DDE is a legacy communications protocol, which means it could disappear from the world of Windows with little warning.
7.4 ACTIVEX

As described earlier in this chapter, the component object model is the key to developing applications that work together, even across multiple platforms. ActiveX, like OLE, is based on COM. This section will discuss what ActiveX is, why it was developed, and how it can be used to improve your applications.

7.4.1 DESCRIPTION OF ACTIVEX

ActiveX controls were formerly known as OLE controls or OCX controls. An ActiveX control is a component that can be inserted into a Web page or application in order to reuse the object's functionality programmed by someone else. ActiveX controls were created to improve on Visual Basic extension controls. ActiveX controls provide a way to allow the tools and applications used on the Web to be integrated together.

The greatest benefit of ActiveX controls is the ability to reduce development time, and to enhance Internet applications. With thousands of reusable controls available, a developer does not have to start from scratch. The controls available to the developer also aid in increased functionality. Some controls that have already been developed will allow the developer to add options to the Web site without having to know how to implement functions. The ability to view movie files, PDF files, and similar interactive applications is made possible through the use of ActiveX.

ActiveX is currently supported in the Windows 95/98/NT platforms, as well as Web browsers for UNIX and the Mac. ActiveX, which is built on COM, is not Win32-specific. This provides the ability to be cross-platform compatible, making ActiveX available to the widest possible group of users.

ActiveX controls can be developed in a number of programming languages, including Microsoft Visual Basic and Visual C++. The key to compatibility is the COM standard that ActiveX is built with. Since a developer can use the language that is most convenient, the development time is reduced. The programmer will not have to learn a new programming language to develop the control.

Some of the definitions of ActiveX technology have roots in object-oriented (OO) design. COM is not completely OO; however, much of the OOP design methodology is used in COM. The main benefits of OO are Encapsulation, Inheritance, and Polymorphism. For more information on these subjects, see Chapter 10.

7.4.2 ACTIVEX DEFINITIONS

First, we will discuss some of the main ActiveX technologies. The main divisions of ActiveX include automation, ActiveX documents, ActiveX controls, and ActiveX scripting. After the discussion of these technologies, we will discuss some of the terms used with ActiveX as well as COM. These terms include properties, methods, events, containers, persistence, servers, clients, linking, and embedding.
7.4.2.1 ActiveX Technologies

ActiveX automation is one of the most important functions of ActiveX. Automation is the ability of one program to control another by using its methods and properties. Automation can also be defined as the standard function that allows an object to define its properties, methods, and types, as well as provide access to these items. The automation interface, Idispatch, provides a means to expose the properties and methods to the outside world. An application can access these items through its “Invoke” method. Programs being able to work together is critical to software reuse. Automation allows the user to integrate different software applications seamlessly.

ActiveX documents (previously called OLE documents) are the part of ActiveX that is involved in linking, embedding, and editing objects. ActiveX documents deals with specific issues relating to "document-centric" functionality. One example is in-place editing. When a user wants to edit an Excel spreadsheet that is embedded in a Word document, the user double-clicks on the spreadsheet. Unlike previous versions of OLE documents, the spreadsheet is not opened in Excel for editing. Instead, Word and Excel negotiate which items in the toolbar and menus are needed to perform the editing. This function of ActiveX allows the user to edit the sheet in Word while still having all the necessary functionality. Another example of ActiveX documents is Web page viewing. When someone viewing a Web page opens a file, like a PDF file, the file can be displayed in the Web browser without having to save the file and open it separately in a PDF reader.

ActiveX controls (which replace OCX controls) are reusable objects that can be controlled by a variety of programming languages to provide added functionality, including support for events. If you listen to the Microsoft information, ActiveX controls are designed for the World Wide Web; however, ActiveX controls can be used in other languages, including Visual Basic and LabVIEW (Version 5.0 or later).

ActiveX scripting is a means to drive an ActiveX control. This is mainly used in Web page development. An ActiveX control is lifeless without code to operate it. Since code cannot be embedded in the Web page, another method of control is necessary. That method is scripting languages. There are two common scripting languages supported in Web pages that are ActiveX compliant. The first is JScript (a type of JavaScript), and Microsoft Visual Basic Scripting Edition (VBScript).

7.4.2.2 ActiveX Terminology

Simply put, a method is a request to perform a function. Let’s say we are programming a baseball team. The baseball team is made up of players. A pitcher is a specific type of player. The pitcher must have a ThrowBall method. This method would describe how to throw the ball. A full description is included at the beginning of this chapter.

A property is the definition of a specific object's parameters or attributes. For instance, in our baseball example, the player would have a uniform property. The user would be able to define whether the player was wearing the home or road uniform.
An event is a function call from an object that something has occurred. To continue the baseball analogy, an event could be compared to the scoreboard. When a pitch is made, the result of the pitch is recorded on the scoreboard. The scoreboard will show ball or strike depending on the event that occurred. Events, as well as methods and properties, occur through automation mechanisms. Events are covered in more detail in the following section.

A container is an application in which an object is embedded. In the example of an Excel spreadsheet that is embedded in a Word document, Microsoft Word is the container. LabVIEW is capable of being a container as well.

When an object is linked in a container, the object remains in another file. The link in the container is a reference to the filename where the object is stored. The container is updated when the object is updated. The benefit of linking is the ability to link the same object in multiple files. When the object is updated, all of the files that are linked to the object are updated.

When an object is embedded in a container, the object's data is stored in the same file as the container. If an Excel worksheet is embedded in a Word document, that data is not available to any other application. When the worksheet is edited, the changes are saved in the data stream within the Word document. With both linking and embedding, a visual representation of the data in the container is stored in the container's file. This representation, called presentation data, is displayed when the object is not active. This is an important feature because it allows the file to be viewed and printed without having to load the actual data from the file or data stream. A new image of the data is stored after the object has been edited.

Persistence is the ability of a control to store information about itself. When a control is loaded, it reads its persistent data using the IPersistStorage interface. Persistence allows a control to start in a known state, perhaps the previous state, and restores any ambient properties. An ambient property is a value that is loaded to tell the control where to start. Examples of ambient properties are default font and default color.

### 7.4.3 Events

An event is an asynchronous notification from the control to the container. There are four types of events: Request events, Before events, After events, and Do events. The Request event is when the control asks the container for permission to perform an action. The Request event contains a pointer to a Boolean variable. This variable allows the container to deny permission to the control. The Before event is sent by the control prior to performing an action. This allows the container to perform any tasks prior to the action occurring. The Before event is not cancelable. The After event is sent by the control to the container indicating an action has occurred. An example of this is a mouse-click event. The After event allows the container to perform an action based on the event that has occurred. The final event is the Do event. The Do event is a message from the container to the control instructing it to perform an action before a default action is executed.

There are a number of standard ActiveX control events that have been defined. Some standard events include Click, DblClick, Error, and MouseMove. These events
have Dispatch Identifications and descriptions associated with them. DispIDs have both a number and name associated with them to make each event, property, and method unique. Microsoft products use the dispatch ID numbers, where LabVIEW uses the dispatch ID names. The standard events have been given negative DispIDs.

### 7.4.4 Containers

An ActiveX container is a container that supports ActiveX controls and can use the control in its own windows or dialogs. An ActiveX control cannot exist alone. The control must be placed in a container. The container is the host application for an ActiveX control. The container can then communicate with the ActiveX control using the COM interfaces. While a number of properties are provided, a container should not expect a control to support anything more than the IUnknown interface. The container must provide support for embedded objects from in-process servers, in-place activation, and event handling. In addition to the container providing a way for the application to communicate with the ActiveX control, the container can also provide a number of additional properties to the control. These properties include extender properties and ambient properties.

The container provides extender properties, methods, and events. They are written to be extensions of the control. The developer using the control should not be able to tell the difference between an extender property and the control’s actual property. There are a few suggested extender properties that all containers should implement. These controls are Name, Visible, Parent, Cancel, and Default. The Name property is the name the user assigns to the control. The Visible property indicates whether the control is visible. The Parent property indicates what the parent of the control is. The Cancel property indicates if the control is the cancel button for the container. The Default property indicates if the control is the default button for the container.

There are a number of additional extender properties that are available to be used. Ambient properties are “hints” the container gives the control with respect to display. An example of an ambient property is the background color. The container tells the control what its background color is so the control can try to match. Some of the most used ambient properties are UserMode, LocaleID, DisplayName, ForeColor, BackColor, Font, and TextAlign. The UserMode property defines whether the control is executing at run time or design time. The LocaleID is used to determine where the control is being used. This is mainly for use with international controls. The DisplayName property defines the name set for the control. The ForeColor and BackColor define the color attributes for matching the control’s appearance to the container.

### 7.4.5 How ActiveX Controls Are Used

The first requirement for using an ActiveX control or a COM object is that it be registered in the system. When applications like Microsoft Word are installed on a computer, the installer registers all of the COM objects in the system registry. Each time the application is started, the information in the registry is verified. There is a slightly different mechanism when the COM object or ActiveX control is contained.
in a Dynamic Link Library (DLL). The specifics will not be mentioned here; the important concept is that the item is known in the system.

The next issue is the interfaces to the control being used. The program using the control needs to be able to access the controls interfaces. Since the ActiveX control is built using COM objects, the interfaces are the common interfaces to the COM object. Some of the most common interfaces are mentioned in the section on COM. These include IDispatch and IUnknown. Most of the interfaces can be created automatically by programming tools without having to do the coding directly.

### 7.5 LABVIEW AND ACTIVEX

The previous sections discussed OLE, COM, and ActiveX to give you a better understanding of how they are used. The terminology, evolution, significance, and operation of these technologies were explained to give you the needed background for using ActiveX effectively. This section will go one step further to define how LabVIEW works with ActiveX. This includes discussions of how ActiveX can be used in LabVIEW, the ActiveX container, and the functions available for use in the code diagram. The goal of this section is to show you LabVIEW’s ActiveX interfaces by describing the tools that are accessible to programmers. This chapter will conclude with a brief review of the VI Server and the related functions. The ActiveX and VI Server functions are very similar, making it an appropriate topic to include in this chapter.

The next chapter provides numerous ActiveX examples to demonstrate how to utilize this feature in LabVIEW programs. It opens the door to a whole new set of possibilities that you may not have thought about before. The examples will provide the foundation for successfully implementing ActiveX through the illustration of applications that are practical.

#### 7.5.1 THE LABVIEW ACTIVEX CONTAINER

The ability to act as an ActiveX control container was first added to LabVIEW 5.0. By adhering to specifications for interacting with ActiveX controls, LabVIEW has become a client for these controls. It allows LabVIEW to utilize the functionality that the controls have to offer. To the user, an ActiveX control appears as any other control on the user interface. The operation of the control is seamless and unnoticeable to the end user. When used in conjunction with the available LabVIEW controls and indicators on the front panel, it becomes an integrated part of the application.

#### 7.5.1.1 Embedding Objects

LabVIEW allows ActiveX controls and documents to be embedded on the front panel of a VI by using the container. Once the control or document has been placed inside the container, it is ready for use and can be activated in place. Additionally, the objects’ properties, methods, and events (supported in LabVIEW 5.1 or later) are made available to the programmer for use in the application.

A container can be placed on the front panel by accessing the ActiveX subpalette from the Controls palette. Then, to drop a control or document inside the container,
pop up on the container and select *Insert ActiveX Object* from the menu. Figure 7.1 displays the dialog box that appears with the options that are available to the programmer once this is selected. Three options are presented in the drop down box: Create Control, Create Document, and Create Object from File. The first option is used to drop a control, such as a checkbox or slide control, into the container. The other two are used to drop ActiveX documents into the container.

When Create Control is selected in the drop-down box, a list of ActiveX controls along with a checkbox, Validate Servers, will appear. The list consists of all ActiveX controls found in the system registry when the servers are not validated. If the box is checked, only registered controls that LabVIEW can interface to will be listed. For instance, some controls that come with software packages that you purchase may have only one license for a specific client. Any other client will not have access to this control’s interfaces. Sometimes third-party controls do not adhere to ActiveX and COM standards. LabVIEW’s container may not support these interfaces, which prevents it from utilizing their services. After a registered control is dropped into the container, its properties, methods, and events can be utilized programmatically. A refnum for the container will appear on the code diagram.

Selecting Create Document also produces a list of registered document types that can be embedded on the front panel. These are applications that expose their services through an interface, allowing LabVIEW to become a client. When Create Object from File is selected, the user must select an existing file to embed on the front panel with an option to link to it directly.

In both cases, the embedded object behaves as a compound document. Compound documents were described earlier in the OLE section of this chapter. Double-clicking on the object launches the associated application. The user can then edit the document using the application. The automation interfaces that are supported by the application can be utilized from the code diagram once the object is placed inside the container. Once again, a refnum representing the container will appear on the code diagram. If Link to File is checked, the embedded object will reflect any
modifications that are made directly to the file. This functionality is referred to as “OLE Automation.”

### 7.5.1.2 Inserting ActiveX Controls and Documents

This section will demonstrate how to insert and use ActiveX documents and controls within LabVIEW’s container. Figure 7.2 shows the front panel of a VI with the Microsoft Slider Control, Version 5.0 (SP2), inserted into the container following the procedure described in the previous section. This control is similar to LabVIEW’s built-in Horizontal Slide control.

The properties of the slide control can be modified by right-clicking on it and selecting Properties under the Slider submenu. Figure 7.3 illustrates the window that appears for setting the slide’s properties. The Appearance tab allows the programmer to alter the orientation, tick style, tick frequency, and mouse style of the control. Essentially, every ActiveX control defines properties for which you can either set or retrieve the current value.

Figure 7.4 displays the code diagram of the same VI. Note that the ActiveX control appears as a refnum on the diagram. This is a simple example in which the Property node, detailed further in Section 7.5.2.2, is used to retrieve the value of the control set from the front panel. With LabVIEW’s ActiveX container, numerous other controls can be integrated into an application. The programmer is no longer limited to the built-in controls.

ActiveX documents are embedded into LabVIEW’s container by either inserting an existing file or creating a new document with an application that exposes its services. Figure 7.5 illustrates the front panel of a VI with a Microsoft Excel worksheet dropped into the ActiveX container. This was done by choosing Create Document from the drop-down menu, then selecting Microsoft Excel Worksheet from the list of available servers. The text shown was entered in the cell by double-clicking on the worksheet to launch Microsoft Excel. Alternatively, you can pop up on the container worksheet and select Edit from the Worksheet submenu. Figure 7.6 shows the same VI, but the Excel sheet is white, the way you are probably used to looking at it. This was done by simply using the paintbrush to color the embedded object. The container refnum can now be utilized programmatically to interface with the worksheet’s properties and methods.

### 7.5.2 The ActiveX/OLE Palette

The Functions palette holds the ActiveX/OLE subpalette (ActiveX subpalette in LabVIEW 5.1). This subpalette contains the automation functions that can be used to interface with ActiveX servers. This section briefly describes these functions: Automation Open, Automation Close, Invoke Node, Property Node, and (Variant) to G Data. The OLE Variant control will also be explained. With these functions, you have everything you need to work with and utilize ActiveX within LabVIEW. You really do not need to know the details of how COM works to use the services provided by servers. It is similar to using GPIB Read or Write in an application.
These functions provide a layer of abstraction so programmers do not have to learn the intricacies that are involved.
7.5.2.1 Automation Open and Close

The function of Automation Open is to open a refnum to an ActiveX server. This refnum can then be passed to the other functions in the palette to perform specific actions programatically. To create the Automation Refnum, first pop up on the Automation Open VI to choose the ActiveX class. Select *Browse* in the Select
ActiveX Class submenu to see a list of the available controls, objects, and interfaces that LabVIEW has found on the system. Figure 7.7 shows the dialog box that appears with the drop-down menu for the type library, and the selection box for the object. LabVIEW gets information on the server’s objects, methods, properties, and events through the type libraries. Additionally, a *Browse* button lets the programmer find other type libraries, object libraries, and ActiveX controls that are not listed in the drop-down box.

Once you have found the object that you want to perform automation functions on, select *OK* on the dialog box and the refnum will be created and wired to Automation Open. If you wire in a machine name, a reference to an object on a remote computer will be opened using DCOM. If it is left unwired, the reference will point to the object locally. DCOM objects can only be instantiated from the Automation Open VI. LabVIEW 5.0 does not have the option to wire-in the machine name. Only Version 5.1 or later can utilize DCOM to open references to objects on remote computers. DCOM must be installed with LabVIEW to be able to use remote objects on Windows 95.

As the name suggests, Automation Close is used to close an automation refnum. You should always remember to close refnums when they will not be used further or before termination of the application in order to deallocate system resources.

### 7.5.2.2 The Property Node

The Property node is used to get or set properties of an ActiveX object. A refnum must be passed to the Property node to perform the Get or Set action on the object. The automation refnum, either from Automation Open or the refnum created from inserting a control into the front panel container, can be passed to the node. Note that this is the same Property node that is available in the Application Control subpalette to read or write properties for an application or VI. When performing application control, however, a reference must first be opened to an application or VI using the Open Application Reference function.
Once a refnum has been passed to the Property node, you can pop up on the node and select a property that you wish to perform a read or write action on. The list of properties the object supports will be listed in the Properties submenu. If you wish to get or set more than one property, simply add elements from the pop-up menu, or drag the node to include as many as are needed. Then, select a property for each element. To change, or toggle, the action from read to write (or vice versa), select the associated menu item by popping up on the node. Some properties are read only, so you may not be able to set these properties. In this case, the selection in the pop-up menu will be disabled.

Figure 7.8 is an example in which the Property node is used to get the default file path that Microsoft Excel uses. Automation Open is used to open a reference to Microsoft Excel; Microsoft Excel 8.0 Object Library Version 1.2 was selected from the type library drop-down box, and Application (Excel.Application.8) from the object list. DefaultFilePath is one of the items that is available for the application in the Properties submenu of the node. C:\My Documents was returned as the default file path that Excel uses when Save or Open is selected from the File menu. Alternatively, this property can easily be used to set the default file path that you want Excel to look in first when Open or Save is selected. First, you must pop up on the Property node and select Change to Write from the menu. Then a file path can be wired into the node. This is depicted in Figure 7.9, where C:\National Instruments is set as the default path.

Figure 7.9
7.5.2.3 The Invoke Node

The Invoke node is the function used to execute methods that an ActiveX control makes available to a client. The number of methods that a particular control offers can vary depending on its complexity. Simple controls such as the Microsoft Slider Control, shown earlier in Figure 7.2, has only five methods. Complex objects like Microsoft Excel, may have upwards of several hundred methods available. Virtually all actions that can be performed using Microsoft Excel can also be performed using automation.

An automation refnum, either created from a control placed in the front panel container or from Automation Open, must be passed to Invoke node. Once an automation refnum has been wired to Invoke node, you can pop up on it and select an action from the Methods submenu. The method selected may have input or output parameters that must be considered. Some inputs are required while others are optional. The data from output parameters can be wired to indicators or used for other purposes as part of the code.

Figure 7.10 shows the code diagram of an example using the Invoke node to execute a method in Microsoft Excel. As before, Automation Open is used to open a reference to Microsoft Excel. Then, the refnum is passed to the Invoke node, and CheckSpelling is selected from the Methods submenu. When the method is selected, the input and output parameters appear below it. The CheckSpelling method checks the spelling of the word passed to it using Excel’s dictionary. If the spelling is correct, a “true” value is returned. If the spelling is incorrect, a “false” value is returned. The only required input for this method is Word; the rest are optional. In a similar manner, you can utilize the services offered by any object that supports automation. This is a very effective way to realize code reuse in applications.

Complex ActiveX servers arrange their accessible properties and methods in a hierarchy. This requires the programmer to use properties and methods of a server to get to other available properties and methods. Applications such as Microsoft Excel and Word operate in a similar manner. After you have opened a reference to one of these applications, the Invoke node and Property node can be used to get to other properties and methods that are not directly available. The first reference opened with Automation Open is also known as a “createable object.” Note that when you are selecting the ActiveX class for Automation Open, the dialog box (shown in Figure 7.7) gives you the option to show only the createable objects in the selection list. When the option is enabled, LabVIEW lists those objects that are at the top of the hierarchy. What does this mean to a programmer who wants to use ActiveX and automation? For simple objects, a programmer is not required to know the details about the hierarchy to use them. They have relatively few properties and methods, and are simple to use programmatically. When complex objects or applications are being used, however, a programmer needs to know how the hierarchy of services is arranged to get the desired result in a program. This means that you have to refer to the documentation on the server, or help files, to use them effectively. The documentation will help guide you in utilizing ActiveX in your applications.
Figure 7.11 illustrates the hierarchy of Microsoft Word through an example. This VI opens a Microsoft Word document, named test.doc, and returns the number of total characters in the document. First, a reference to Microsoft Word Application is opened using Automation Open. Then, Documents is the first property selected with the Property node. Documents is a refnum with a set of properties and methods under its hierarchy. The Open method is then used to open the file by specifying its path. Next, Characters is a property whose Count property is executed to retrieve the number of characters contained in the document. Observe that all references are closed after they are no longer needed. The Microsoft Word hierarchy used in this example proceeds as shown in Figure 7.12. Microsoft Word application is at the top of the hierarchy with its properties and methods following in the order shown.

**FIGURE 7.11**

**FIGURE 7.12**
7.5.2.4 To G Data (Variants)

A variant is a data type that varies to fit whatever form is required of it. A variant can represent strings, integers, floating points, dates, currency, and other types, adjusting its size as needed. Variants are explained in more detail in Section 7.2.1. This data type does not exist within LabVIEW; however, many ActiveX controls do make use of it. Variants are valid data types in Visual Basic, which is often used to create ActiveX controls. Therefore, with the addition of ActiveX functionality, LabVIEW must be able to deal with variants in order to pass and retrieve data with objects.

LabVIEW supplies a control and a function VI to handle variants when working with ActiveX because it does not interpret variants. The OLE Variant is a control available in the ActiveX subpalette provided to facilitate passing variant data types to servers. The To G Data function converts variant data to a valid data type that LabVIEW can handle and display programmatically. To use this function, simply wire in the Variant data to be converted and the type that you want it converted to. A table of valid data types can be accessed from the on-line help by popping up on the function. You can wire in any constant value among the valid data types to which you need the Variant converted.

Figure 7.13 shows an example using To G Data. The code diagram shown is a subVI in which the value of the active cell in a Microsoft Excel workbook is being read. Its caller, which opens a reference to Excel and the workbook, also passes in the Application refnum for use. As shown, the data type for the cell value is a variant. The actual data in the spreadsheet is a string, therefore, the variant passed from Excel must be converted to a valid LabVIEW data type using To G Data. An empty string constant is wired to the type input and a string indicator is wired to its output. If the data in the workbook is a number, a numeric constant (integer or floating point) can be wired to the type input for conversion.

When an ActiveX object requires a variant data type as input, the programmer is usually not required to perform any conversion from the valid LabVIEW types. Section 7.2.1 discusses Unicode and possible conversion problems of data types between LabVIEW and controls. Look back at Figure 7.11, the example in which a Microsoft Word document was opened and the number of characters retrieved. The Open method required a file name of variant type. A string constant was wired to this input without any conversion. If you look closely, you will see a coercion dot, indicating that LabVIEW coerced the data to fit the type required.
7.5.2.5 Using the Container versus Automation

When you first begin to work with ActiveX in LabVIEW, it may be unclear whether to use the front panel container or Automation Open to utilize the services of an object. For instance, some servers can either be used by dropping them into the container or by creating a refnum with Automation Open. The general rule of thumb, when using ActiveX is to utilize the front panel container when working with controls or embedded documents that the user needs to view. The CW Knob Control (evaluation copy that is installed with LabVIEW 5.0) is an example of a control that needs to be placed in the front panel container so that the user can set its value. If Automation Open is used to create the refnum for the knob, the user will not be able to set its value. This applies to controls, indicators, or documents that the user must be able to view or manipulate in an application. Once placed in the container, the Invoke node and Property node can be used to perform necessary actions programmatically, as demonstrated in previous examples.

When you need to use ActiveX automation to work with applications like Microsoft Word or Excel, it should be done from the code diagram, without using the container. Use Automation Open to create a refnum by selecting the ActiveX Class to make use of the available services. Then, use Invoke node and Property node to perform needed actions, as shown in Figure 7.11. A front panel container was not used in that example. You may need to refer to documentation on the objects’ hierarchy of services. On the other hand, if a specific document’s contents need to be displayed and edited, then you can embed the document on the front panel and perform the actions needed.

7.5.3 Event Support in LabVIEW 5.1

LabVIEW 5.1 is the first version that supports ActiveX event handling. Section 7.4.3 described the various events in detail. Many ActiveX controls define a set of events, in addition to properties and methods that are associated with it. These events are then passed to the client, or container, when they occur. When a specific event is passed back, the client can deal with the event by executing code to perform any necessary actions. This is significant because LabVIEW can now interface to properties, methods, and events to fully utilize all of the services that ActiveX objects have to offer. This section will explain and demonstrate how to use the functions available in the ActiveX Events subpalette.

7.5.3.1 List Event Descriptions

In order to be able to use the events that a control passes back to LabVIEW, the programmer needs to know all of the events that a control supports. Documentation or reference material for the control must be consulted to determine the events that are supported. Alternatively, List Event Descriptions.vi can be used to retrieve the supported events from a control. This VI reads the event information from the control’s type library and retrieves it for use by the programmer.
Figure 7.14 shows the information that is returned from the List Event Descriptions.vi. The Microsoft Forms 2.0 CommandButton was used in this example. After the control was placed in the front panel container, the refnum was passed to this VI. An array of event descriptions supported by the control was returned as shown in the illustration. These events, upon occurrence, are passed back to LabVIEW for use programmatically.

7.5.3.2 ActiveX Event Queues

ActiveX event queues are used to retrieve the occurrence of a specific event from a control. The purpose of Create ActiveX Event Queue.vi is to create this queue which holds instances of the event. The automation refnum and the event name of the queue to be created must be wired in to the VI. Consult the documentation for the event names available for the controls. A Wait option is also provided for the queue. A Non Queued Mode captures a single instance of the event, while the Queued Mode stores multiple occurrences of the event.

The event queue created for a particular event, using Create ActiveX Event Queue.vi, can then be passed to Wait on ActiveX Event.vi. Wait on ActiveX Event.vi waits for an occurrence of the event for which the queue is created, or until the wait time is exceeded. It passes out a cluster of Event Data that contains the Event Name, Event ID, and the associated parameters for the event and their data.

Figure 7.15 is an example of how these two VIs are employed to take advantage of ActiveX events in LabVIEW 5.1. Once again, the Microsoft Forms 2.0 CommandButton is used as the front panel ActiveX control. First, the Caption property for the button is modified to describe the button for the user. The string “Play Sound
File” is passed to the Property node to modify the caption. Then, a queue is created for the Click event, which is one of the events supported by the control. Creating the event queue notifies the ActiveX control that the client will provide a function to deal with the event. The Wait on ActiveX Event.vi is used to wait for the user to click on the button from the front panel. Once the button is clicked, a sound file is played. This is a simple example of how useful events can be in a program. A polling loop would have to be used to determine if the button was pressed if event capability did not exist. Events are more efficient because the code “sleeps” until the event occurs. Figure 7.16 displays the front panel of the VI. The ActiveX control is shown along with the Event Data cluster, which shows the information that is returned by the Wait on ActiveX Event.vi. This information can be used programmatically as needed to perform other functions.

ActiveX controls usually have support for more than one event that can be used in an application. The CommandButton, for example, supports 11 different events. Queues need to be created for all events you want to support. When you want to wait for an instance of any one event among several, Wait on ActiveX Event from Multiple.vi can be used. Create an event queue for each event, as shown in the previous example. Then, form an array using these event queues’ output from Create ActiveX Event Queue.vi and wire it in to this VI. The Event Data will be returned as a cluster containing the information described earlier. You can then unbundle the Event Name from the cluster and use this output to drive a case statement. Different actions can be performed depending on the event passed back from the control.

Destroy ActiveX Event Queue.vi destroys the event queue passed to it, as the name suggests. Any other functions that are waiting when the event queue is destroyed are aborted. If you wish to clear event data from a queue instead, Clear ActiveX Event Data.vi is also available in the subpalette. Simply wire in the queue that you need cleared to this VI. You should flush and destroy the event queues when you are finished using them.

### 7.5.4 LabVIEW as ActiveX Server

LabVIEW has both ActiveX client and server capabilities. This means that you can call and use external services inside LabVIEW’s environment, as well as call and use LabVIEW services from other client environments such as Visual Basic. This functionality enhances the ability to reuse LabVIEW code, by being able to call and run VIs from other programming languages. It also gives you flexibility to use the language that best suits your needs without having to worry about code compatibility.
When using LabVIEW as an ActiveX server, the Application and Virtual Instrument classes are available to the client. Consult the on-line reference for details on the methods and properties that these classes expose to clients. To use LabVIEW as an ActiveX server, you must first ensure that this functionality has been initiated. Select Preferences from the Edit menu, and VI Server: Configuration from the dropdown box as shown in Figure 7.17. Check the ActiveX protocol selection and the appropriate Server Resources as needed. You can allow VI calls, VI methods and properties, and Application methods and properties. The on-line reference also has an ActiveX example to demonstrate how this feature can be used. It is an example
of a Visual Basic macro in which a VI is loaded, run, and its results retrieved for further use.

7.6 THE VI SERVER

VI Server functionality allows programmers to control LabVIEW applications and VIs remotely. The functions available for use with the VI Server are on the Application Control subpalette. You must first ensure that LabVIEW is configured correctly to be able to use this capability. Figure 7.17 displays the VI Server configuration dialog box. Activate the TCP/IP protocol, then enter the port and enable the appropriate server resources you wish to use. In the VI Server: TCP/IP menu item, you can enter a list of computers and IP addresses that you wish to allow or deny access to the VI Server. Finally, the VI Server: Exported VIs item allows you to create a list of VIs that you want to allow or deny access to remotely. The wildcard character (*) allows or restricts access to all VIs on the machine.

The Open Application Reference function is used to open a reference to LabVIEW on a remote machine. The machine name, or IP address, must be wired to the function along with a TCP port number. This should coincide with the TCP port number that was specified in the configuration for VI Server on the remote machine. If a machine name is not wired to the function, a local reference to LabVIEW will be opened. Once the reference is opened, the programmer can manipulate the properties and methods of the local or remote LabVIEW application. Use Close Application or VI Reference when you are finished using the reference in an application.

The Property node is used to get or set properties and the Invoke node is used to execute any methods. These functions are identical to the ones described previously while discussing ActiveX functions. The Property node and Invoke node from both ActiveX and Application Control subpalettes can be used interchangeably.

The code diagram in Figure 7.18 demonstrates how to make use of the VI Server. This example simply retrieves the version number of the LabVIEW application on a remote computer. First, a reference to the application is opened and the machine name and port number are wired in. Then the Property node is used to get the version number, and the reference is subsequently closed. The on-line help describes all of the properties and methods that can be utilized programmatically.

FIGURE 7.18
The Open VI Reference function is used to open a reference to a specific VI on either a local or remote machine. Simply wire in the file path of the VI to which you wish to open the reference. If the VI is on a remote computer, use Open Application Reference first, and wire the refnum to the Open VI Reference input. Figure 7.19 shows an example first introduced in the Exception Handling chapter. It opens a reference to External Handler.vi and sends it the error cluster information using the Set Control Value method. Then the External Handler is executed using the Run VI method. Finally, the reference to the VI is closed with Close Application or VI Reference.

BIBLIOGRAPHY


G Programming Reference, National Instruments