Bitter, Rick et al "ActiveX Examples"

LabVIEW Advanced Programming Techniques

Boca Raton: CRC Press LLC, 2001
This chapter is a follow-up to Chapter 7 on ActiveX. In Chapter 7, you were introduced to OLE, COM, DDE, ActiveX, and their significance to programming. The related terminology was defined along with a description on how these technologies work. The chapter concluded by showing LabVIEW’s ActiveX interface and the related tools available to the programmer for developing applications.

In this chapter we will provide several examples that utilize ActiveX controls and automation. The material here serves two purposes. The first is to give you enough exposure to ActiveX so that you will be comfortable using it in your own applications effectively. By employing the services offered by ActiveX objects, code reuse is significantly enhanced. This is a key advantage that was gained by the introduction of COM. The second intent is to give you some practical examples that you can modify and utilize in your applications. Even if the examples are not directly applicable to your situation, they will give you a new way of thinking to apply ActiveX that you may not have considered before.

8.1 COMMON DIALOG CONTROL

The Common Dialog control is familiar to almost every Visual Basic programmer. Microsoft elected to provide a uniform interface to the dialog boxes for printing, opening, saving, and color selection. These were wrapped into a single dialog box and became known as the Common Dialog control. Every application you use, you can see the familiar Print, Open, and Save Boxes is using this control. Being a user interface element, it was desirable for Microsoft to have standard mechanisms for users to perform common tasks. This allows the Windows operating system to provide a consistent look and feel to end users regardless of which company was developing software to use on the operating system.

We can use the Common Dialog control to keep the Windows look and feel consistent for our end users. This control is useful if you are using file saving and not using the high-level interfaces provided by LabVIEW. If you are using the high-level interfaces, you do not need to use this control; LabVIEW is using it for you!

This example uses the Microsoft Common Dialog Control Version 6.0. The Common Dialog Control is relatively simple to use and will serve as the introductory example. It is useful for prompting the operator to select a specific file for opening and saving purposes, while also allowing them to navigate through the Windows directories.
Figure 8.1 displays the code diagram of Common Dialog.vi. The Common Dialog control was placed in the front panel ActiveX container, and is represented by the refnum on the diagram. The objective of this VI is to display the dialog box and instruct the user of the application to select a file. The name of the file that the user selects is then retrieved from the control and used elsewhere. In this example, the first action taken is to set the InitDir property to C:\. This causes the dialog box to display the contents of C:\ when it appears. Next, the DialogTitle property is set to prompt the user to select a specification file. Then the Show Open method is executed, which simply displays the dialog box. Finally, the FileName property is read back to find out the name of the file that was selected. The Common Dialog.vi can be used as a subVI whenever a similar function needs to be performed. Figure 8.2 shows the dialog box as it appears when the VI is executed.

The front panel container was used to insert the Common Dialog Control in this example, but we could just as easily have used Automation Open. The reason for using the container was to have the control displayed instead of the automation refnum. This allows the programmer to modify properties of the control quickly, by popping up on it. On the block diagram, the InitDir and DialogTitle properties could have been set in the same step, by popping up on the Property node and selecting Add Element. Since this is the first example presented, it was done with two separate property nodes to simplify the block diagram.

Figure 8.1

Figure 8.2
8.2 PROGRESS BAR CONTROL

The Microsoft Progress Bar can be used to keep a consistent look and feel in your applications. Programmers have been using slide controls for years to emulate the Microsoft Progress Bar. Now that the control is available through ActiveX, we can use it without any workarounds. As we mentioned, a strong advantage to using this control is that it will always have the same look and feel of other applications running on Windows. In the event that Microsoft releases a new version of Windows containing a new look to the Progress Bar, the change will be automatically updated into your application if you replace the older control with the new one on your system. The way the Progress Bar looks could potentially change with a new release of Windows; however, the interface for this control is not likely to change.

Since this is a user interface element, there are two different methods that we can use to create it: we can insert it into an OLE container, or we can use the ActiveX Open. User interface elements are typically created with an OLE container so there are no issues regarding the location of the control of the display. Therefore, we will not have an open VI for this control; users will need to place an OLE container on the front panel and insert this object. Once the OLE container is placed on the front panel, right-clicking on the control allows you to insert an ActiveX object. Insert a Microsoft Progress Bar control. You may notice that there are several different Progress Bar controls listed with different version numbers. Each release of languages like Visual Basic will have a new version of the Progress Bar control. Typically, we use the “latest and greatest” version of each control, but you may feel free to use any of the versions currently listed. According to the rules of COM, each version of the control should have the same interface. Additional functionality can only be added to new interfaces. This ensures that you will not have compatibility issues if another version of the control is updated on your system.

Once the control is placed on the front panel, you can resize it to dimensions that are appropriate for your application. The control itself has 14 properties and 3 methods. The methods are About Box, OLEDrag, and Refresh. The About box simply displays a window showing the copyright information for this control. Users may be impressed with the quality of the Progress Bar, but not impressed enough to want to see the credits for the control. We will also not need the OLEDrag or Refresh methods. The control is capable of updating itself, and there is no need to manually update it.

Properties for this control include display concerns such as appearance, orientation, mouse information, and border style. If you want the typical 3-D look and feel, the appearance property should be set to 3-D. This control uses enumerated types to make it easy for programmers to determine which values the control will accept. All appearance properties for the control use enumerated types. Orientation allows programmers to have the control move horizontally or vertically. The default value is horizontal, but it is possible to have the Progress Bar move up and down, as is done in Install Shield scripts for hard drive space remaining.

The mouse information allows you to determine which mouse style will be shown when the user locates the mouse over the control. An enumerated type will identify which mouse styles are defined in the system; all we need to do is select one.
Min, Max, and Value properties are used to drive the control itself. Min and Max define the 0% and 100% complete points. These values are double-precision inputs to allow for the most flexibility in the control. Setting the value allows the control to determine what percentage complete the bar should display. This simplifies the task of displaying progress bars over the old technique of using a LabVIEW numerical display type; the calculations are performed for us.

The driver for this control is fairly simple. We will have a few property-setting inputs for the configuration-related properties, and a VI to support setting the Value property. Default values will be assigned to the properties on the VI wrapper so inputs do not need to be applied every time we configure a control. The code diagram for this VI appears in Figure 8.3.

8.3 MICROSOFT CALENDAR CONTROL

This example uses the Microsoft Calendar Control, Version 8.0. In order to use this control you will need to place an ActiveX container on the front panel of your VI. You can select the Calendar control by right-clicking on the container and selecting Insert ActiveX Object. When the dialog box comes up you will need to select Calendar Control 8.0. The properties and methods for this control are very manageable. There are 22 properties and 11 methods. The example used here will exercise a majority of the methods and controls.

The first method displays the About box for the control. There is no input for this method. Actually, none of the methods for the calendar control have inputs. The actions are initiated by simply invoking the methods. There are methods for advancing the calendar control to the next day, week, month, and year. By invoking these methods you will increment the Calendar control by the specified value. These methods will also refresh the calendar display. There are methods for moving the calendar control to the previous day, week, month, and year. There is a separate method for refreshing the calendar control. Finally, there is a method for selecting the current date on the control.

If you were implementing these methods in an application, you could make a front panel selector that would drive a state machine on the code diagram. The state machine would contain a state for each method. The user of the application would be able to interactively move around the calendar through the use of these front panel controls.
The background color is the first property available in the Property node. This input requires a long number representing the RGB color for the control's background color. The valid range of typical RGB colors is 0 to 16,777,215. The next set of properties relate to configuring the day settings. You can read or write the value for the currently selected Day, set the Font, set the FontColor, and the DayLength. The DayLength property designates whether the name of the day in the calendar heading appears in short, medium, or long format. For example, to set the DayLength property to long, you would have to wire a 2 to the property input. This would result in the day being displayed as Monday instead of Mon. or M. The property after DayLength is First Day, which will specify which day is displayed first in the calendar.

The next set of properties relates to the grid style. The GridCellEffect property sets the style of grid lines. They can be flat, raised, or sunken. The GridFont property sets the font for the days of the month. You can also set the font color and line color for the grid. There are five properties relating to visibility. They allow you to show or hide items such as the title. The Value property allows you to select a day on the calendar or read the currently selected date from the calendar. The data type of the Value property is a variant. A Null value corresponds to no day being selected. The ValueIsNull property forces the control to not have data selected. This is useful for clearing user inputs in order to obtain a new value.

In the following example we will configure the Calendar control to use user-selected properties for a number of attributes. The user will be able to set the length of the day and month, as well as the grid attributes. After configuring the display settings, we will prompt the user to select a date. Using ActiveX events, the program waits for the user to click on a date. After the date has been selected, we read in the value of the control and convert the value to the date in days. The program will then calculate the current date in days. The current date will be subtracted from the selected date to calculate the number of days until the specified date. This value will be displayed on the front panel. The code diagram and front panel for this example are shown in Figure 8.4. The day, month, and year are all returned as integers. The month needs to be converted to a string by performing a Pick Line and Append function. This function selects a string from a multiline string based on the index.

8.4 WEB BROWSER CONTROL

This example utilizes the Microsoft Web Browser control. The Web Browser control can be used to embed a browser into the front panel of a VI, and has various applications. This control must be dropped into LabVIEW’s front panel container. It is part of a collection of createable objects included with Microsoft Internet Controls Version 1.1. The Web Browser control corresponds to the Internet Explorer (InternetExplorer.Application.1) object. It allows you to navigate the Web programmatically through the COM interface and see the pages displayed on the front panel window.

The Web Browser control can be very useful as part of an application's user interface. This example will illustrate how to utilize the control in a user interface to display on-line documentation or technical support for an operator. A simplified
user interface is displayed in Figure 8.5 that shows an enumerated control and the Microsoft Forms 2.0 CommandButton. Obviously, this would be only one control among several that you may want to make available in your user interface. The enumerated control lets the operator select the type of on-line support that is desired. When the \textit{Go!!} button is pressed, another window appears with the Web page selected from the control. This is ideal for use in a company’s Intranet, where the Web-based documentation is created and saved on a Web server.

Figure 8.6 shows the code diagram of this user interface. After setting the Caption property of the CommandButton, an event queue is created for Click. The VI sleeps until \textit{Go!!} is clicked, after which Web Browser.vi executes. The Web Browser VI was configured to display the front panel when a call is made to it. This option is made available if you pop up on the icon in the upper right corner of a VI panel or
diagram. Select VI Setup from the pop-up menu and the window shown in Figure 8.7 appears. Note that checkboxes have been selected to show the front panel when the VI is called and to close it after execution if it was originally closed.

The front panel window with the embedded browser appears loading the URL specified, as shown in Figure 8.8. National Instruments’ home page is displayed in the browser window in this example. The Browser control allows the user to click on and follow the links to jump to different pages. When the operator is finished navigating around the documentation on-line, the Done CommandButton is pressed to return to the main user interface.

Figure 8.9 illustrates the code diagram of the Web Browser VI. Based on the support type selected from the user interface, the appropriate URL is passed to the Navigate method on the Browser control. As the front panel is displayed when the VI is called, the URL will begin loading in the browser window. Once again, the Caption property is modified, this time to display “Done” on the CommandButton. Next, an event queue is created and the VI waits for the Click event. When Done is clicked, the queue is destroyed and the window closes, returning to the user interface.
The Browser control does not have many methods or properties in its hierarchy of services, making it relatively simple to use. The CommandButton is also a simple control to use in an application. The button’s support of several ActiveX events makes it an attractive control for use. The Click event is convenient because it eliminates the need for a polling loop.

FIGURE 8.7

FIGURE 8.8
The scripting control is a unique control that was used to add functionality to both Visual Basic and Visual Basic script. One of the internal components of this control is the dictionary. The dictionary is very useful for storing and retrieving data via a key. When working with objects such as clusters that store configuration information for an application, it may be desired to access the clusters by a key. The key can be a number such as an array index. It might be desirable to use a string for the key. Humans tend to identify strings better than numbers, which is why enumerated types are popular in every programming language.

As an example of why a dictionary might be useful, consider a very simple application that stores basic information about people, such as their names, phone numbers, and e-mail addresses. We tend to remember our friends’ names, but their phone numbers and e-mail addresses might elude us. We want to be able to enter the name of one of our friends and have their name and e-mail address returned to us.

As an example of why a dictionary might be useful, consider a very simple application that stores basic information about people, such as their names, phone numbers, and e-mail addresses. We tend to remember our friends’ names, but their phone numbers and e-mail addresses might elude us. We want to be able to enter the name of one of our friends and have their name and e-mail address returned to us.

One solution is to store all the names and addresses in a two-dimensional array. This array would store names, phone numbers, and e-mail addresses in columns. Each row of the array represents an individual. Implementing this solution requires that we search the first column of the array for a name and take the row if we find a match. Elegance is not present, and this approach has problems in languages such as C, where array boundaries do not just grow when we exceed the bounds. Also, consider that if we have a lot of friends and add one past the array boundaries, the entire array will be redimensioned, which will cause performance degradation. The big array solution does have a benefit; it can be easily exported to a tab-delimited text file for presentation and sorting in an application like Microsoft Excel.

It would be easier if we could use a cluster to store all the information, and search the clusters by the name to find the information. We can use clusters and a one-dimensional array, but we still have the problem of arrays and resizing. The next problem we encounter is when we want to remove people from the array. This involves splitting the array and rebuilding it without the undesired row. Memory hits will again show up because we need to allocate two subarrays and one new array.

Linked lists are a possible solution, but searching the list is not particularly efficient. We would have to search the elements one at a time, from beginning to end, or use sorting algorithms to implement other search patterns. Regardless of the
method chosen, we will spend time implementing code or will waste CPU time executing the code. Enter the scripting control and its Dictionary component. Microsoft describes the dictionary as an ActiveX equivalent to a PERL associative array. Associative arrays take two values: one value you want to store, and a value that you want to reference it with. We are not being very specific about the values that are stored and used for reference because the item types are very flexible. If we want to store clusters of information and refer to them by strings, we can. If we are interested in referring to strings by numbers, we can do that, too.

The interface to Dictionary is fairly simple, and a small collection of VIs to use the control are provided on the companion CD. First, we need an open Dictionary VI to create a reference to the ActiveX control. This VI will not be documented because it is a simple automation open command wrapped in a VI. A Close VI is provided; it is an automation close in a wrapper.

To insert a pair of items, called the “key and item,” we invoke the Add Item method. This method is wrapped in a VI for ease of use. The control accepts variants, and we will use the variants with this control. The other option is to have wrappers that accept each data type and polymorph the LabVIEW type into a variant. This would require a lot of work, and the benefits are not very high. Either way, the types need to be polymorphed. Each cluster of personal information we want to insert into the dictionary is added to the map with the Add Item VI. The use of this VI is shown in Figure 8.10.

To verify that an item exists, the Key Exists VI is used. This VI asks the dictionary if a certain key is defined in the dictionary. A Boolean will be returned indicating whether or not the key exists.

Removing items is also very easy to accomplish; the Remove Item VI is invoked. The key value is supplied and the item is purged from the array. Removing all items should be done before the dictionary is closed, or memory leaks will occur. This VI calls the Remove All method and every element in the dictionary is deleted.

If we want to know how many items are in the dictionary, the Count property can be retrieved. Count will tell us how many entries are in the dictionary, but does not give us any information as to what the key names are. A list of all key names can be obtained with the Get Keys VI. This VI will return a variant that needs to be converted to an array of the key type. For example, if you had a dictionary mapping integers to clusters, the output of Get Keys would have to be converted to an array of integers. An example of the conversion is shown in Figure 8.11. Our driver will not convert values for us because this would require a large number of VIs to accomplish, or would not be as reusable if we do not cover all the possible data types. In this case, variant usage is working in our favor.
The array returned by Get Keys can be used with a For loop to get each and every item back out of the dictionary if this is required. We will have an array of elements, which is generally undesirable, but there are cases where we will need to do this. Using the For loop to develop the array is efficient, because auto indexing will allow the For loop to predetermine the size of the output array. This array can then be dumped to a file for storage.

This example demonstrated a simple application of the Dictionary control. This control can be very useful in programming applications where an unknown number of data items needs to be stored for use at a later time. Applications that should consider using this control would have storage requirements for which arrays cannot be practical, and for which using a database would be overkill. Database programming can be challenging, and interfacing to the database may not be very fast. The Dictionary is an intermediate solution to this type of programming power, and variants allow this tool to be very flexible.

8.6 MICROSOFT WINSOCK CONTROL

This section discusses the use of the Microsoft Winsock control. LabVIEW has a native API for TCP and UDP functionality, but LabVIEW’s API does not support events. Event support makes the Winsock control desirable when efficient communications code is needed. LabVIEW’s TCP and UDP APIs do not support events, so receiving data must be done in polling loops. Polling loops are inefficient uses of CPU time. When efficiency is a requirement, the Winsock control can be used for receiving traffic.

Before using this control, we need to verify that a correct resource DLL is available to LabVIEW. The resource directory inside the LabVIEW directory needs to have a DLL named “ax-events.DLL.” The version of this DLL can be determined by right-clicking on the file and selecting Properties. The properties box should have a version tab. The correct version should be 1,0,0,2. If the version number is 1,0,0,1 or earlier, you will experience problems with ActiveX events. This DLL can be downloaded from National Instruments’ Web site, www.natinst.com.

The ActiveX Winsock control is an ActiveX wrapper around the Winsock DLL. LabVIEW’s TCP and UDP VI collection is also a wrapper around this DLL. Winsock is Microsoft’s implementation of the UNIX Sockets standard. Sockets have been a primary means of communications for applications. Without sockets, concepts such as e-mail, WWW, and network news would never have been implemented.
Before we go into the details of using the control, we will present a very brief overview of TCP and UDP. TCP is the Transmission Control Protocol; it is a connection-based protocol. A TCP address is a combination of two numbers: the IP address of the machine and the port (socket) number that is receiving the data. TCP maintains a connection between two computers until either the local or remote machine closes the connection.

User Datagram Protocol (UDP) is a connectionless protocol. UDP does not maintain active connections or provide retry mechanisms. If a packet does not arrive at the remote computer or is corrupted upon arrival, no notification is returned to the sending machine. Despite this limitation, UDP is preferable in applications that maintain numerous connections or have high-speed requirements. It is possible to implement a custom acknowledgment routine where the remote machine would send a reply to each message it received. This seems like a lot of work considering that TCP does this type of work for you. High-speed applications may need to consider the latency in the TCP communications stack. TCP's communication stack is geared to message reliability and there is a fair amount of code needed to perform TCP communications. If an application does not explicitly require notification that a message has arrived, UDP is a lighter and faster protocol.

We will need to develop a driver to use this control; the driver does an excellent job of covering up the ActiveX API with a native set of LabVIEW commands. The driver will also combine a number of common tasks. The first tasks that need to be done at startup are opening the ActiveX refnum and preliminary configurations. As previously stated, the Winsock control supports both UDP and TCP through a single interface. The protocol is selectable with an enumerated type. It stands to reason that the control should be configured with this protocol at startup, this will be a property that is used in the Winsock Open VI. In addition to the protocol, a local port number is not likely to change during the use of this control. Local port will be the second input. If a value not specified, zero will be fed into the control. The control will not let us use a local port value of zero, and will assign a new value for us. The last property that needs to be configured at startup is whether or not we want to bind to the port number. “Bind” informs the operating system that we are reserving that particular port number and no other applications should be allowed access to it. This property is useful for TCP/UDP servers, and will be necessary if a UDP connection is being used. Bind will be given a default value of “false.” The code diagram for the open VI appears in Figure 8.12.

![Code diagram for the open VI](image)

**FIGURE 8.12**
The next method we will need to support in order to use the Winsock control is the Connect method. The Connect method takes a remote host and port number as variant arguments. LabVIEW programmers will likely use a string for the host name and an integer for the port number. There are string-to-IP and IP-to-string functions in the TCP palette for easy conversion between the addressing methods if programmers require them. Connect works differently for TCP and UDP. UDP is a connectionless protocol. It does not make a lot of sense to establish a connectionless link with a connection, but this starts up the control for use with UDP. Connect contacts the remote host and port when the control is using TCP. The common interface makes it easier for programmers to use one control and API for two communications tasks. We will not show the code diagram for this VI; it is available on the companion CD-ROM.

Sending data is a prime purpose for this control, and it supports a method to do just that. Send Data will accept data types of any kind, but our driver example will only support strings. Writing additional wrappers for other data types is a trivial exercise. Send Data should not be invoked unless the Connect method has been called previously. Without the Connect method, a TCP conversation has not been started, and for UDP there is no destination information available. The control will generate an error if you attempt to send data without establishing a connection first.

Receiving data is accomplished with the Get Data method. This method should be used in conjunction with the Data Arrival event. This event will notify an application that data has arrived at the port the control has specified. We will cover Winsock control’s events shortly. In the meantime, the command to retrieve data from the control is Get Data. Get Data supports multiple data types. The three arguments to the Get Data method are data, type, and maximum length. We wired an empty string into the data argument. In languages such as C++ where pointers are used, the control expects the user to supply a location to copy the data. LabVIEW does all of this under the hood, so wiring an empty string supplies a location to use. It is important to understand that LabVIEW quietly discards this information; you just need to give a location for the ActiveX interface to use. Type is a variant argument that would specify the type of data that should be returned, such as integer, string, or floating-point numbers. The data returned is stored in a variant that can be converted to any of the desired types by the To G Data function. This example will only use string data types, but similar to Send Data, it is a trivial exercise to support other data types with this driver.

### 8.6.1 Using Winsock Control with TCP

Using the Winsock control for TCP communications is not complicated, but the control behaves a bit differently for TCP and UDP communications. This section will outline using the control for TCP. The first property that is optional for TCP connections is Bind; binding is not necessary for TCP connections.

Events such as Connection Request need to be used with TCP server applications. This event is analogous to the TCP Listen VI. The corresponding method to a Connection Request event is the Accept method. Accept informs the Winsock control
that we will grant the connection, and the Deny method informs Winsock that we are not interested in talking to the other system.

### 8.6.2 Using Winsock Control with UDP

The Winsock control uses one API for UDP and TCP programming, but the methods and properties behave a bit differently when programming for UDP communications. First, the Bind command is not optional. Since UDP is connectionless, there must be some mechanism for maintaining the port. Recall that “Binding” means informing the operating system that your application is going to camp out at this particular port number. If you do not issue the Bind command, your application will not receive any data, and the application will not give any reasons why data was not received.

The Connect method is still used to specify the remote host and port, but unlike TCP there is no network traffic sent because there is no true connection being established.

Sending data can be done after the connect is issued; the datagram is transmitted to the destination computer, and there will be no acknowledgment that the message was received. If it is necessary to know that the message was, in fact, received, you could have the destination computer send back a reply indicating that. In high-speed applications or applications that are running over very slow links, this method is preferable because there is significantly less overhead to UDP packets. Both TCP and UDP packets will be sent using Internet Protocol (IP). The IP header is 20 bytes long (possibly longer), and a TCP packet has an additional 20 bytes of overhead. UDP only has 8 bytes of overhead, which makes it much more efficient over slow communications channels. In addition, the TCP call stack uses a lot more overhead to process acknowledgments, where UDP has nothing. High-speed applications benefit because you are using the lightweight protocol.

UDP servers will not use the Connection Request event. The Data Arrival event would be used to handle connection requests; it is a simple matter to identify the remote machine and port number. When processing the data arrival event, examine the properties for remote host and remote port to identify where a datagram came from. Additionally, it is possible to use multiple instances of Winsock to improve the capacity of a server. A Winsock control will be created and will bind to a port number known to all clients. Clients will send a datagram to the server. When the server receives the datagram, it opens another Winsock control and sends a reply back to the client from the new Winsock control. The client can then send traffic to the new, private Winsock control. Throughput can be improved by using multiple ports; this is inherently true for TCP connections because they must have unique ports for each connection.

### 8.6.3 Using Winsock in Client Applications

Client applications are easier to program than server applications, so we will develop a simple client application first. For direct comparison, we will develop a client that requests the current time from a server. This is an ActiveX version of the LabVIEW
example code. The LabVIEW API VI can be found in the following path: C:\Examples\COMM\tcpex.llb\Date Client.vi.

To begin, we will need to open an ActiveX refnum for this control. The Winsock Open VI opens the ActiveX control and configures the local port and binding information. We will not bind to this port; it is not necessary in the client application. No local port will be specified; we will let the operating system select one for us. In client applications it is preferable to use a selected port, which allows multiple instances of the application to be used without creating port conflicts.

Once we have a reference to a Winsock control, we will create an event queue for data arrival. We have written a VI wrapper for this specific event because we intend to use it often. This queue will be used to keep a receiving loop idle until data arrives at the port. Keeping sections of the code diagram idle until necessary minimizes CPU utilization.

Now that we have a reference to the control and data arrival queue, we will issue a connect command to the remote machine. Since this is a TCP application, the Connect routine will contact the remote machine and request a connection. This basically ends half of the application; there is no need to issue any additional commands until the client is ready to terminate.

A While loop will be used to wait on the data arrival event; this allows the application to continuously listen for traffic. A Wait on Event VI will be used to keep the loop idle until data arrives. Once data arrives, the loop will execute the Get Data VI and post the returned string to the date time indicator. A shift register is used to loop back error information. The loop is driven by the Stop Button control, and will exit the loop when a user sets the control value to “false.” Once this loop terminates, the character “z” will be sent to the server to indicate that we would like to close the connection. The code diagram for this application appears in Figure 8.13.

8.6.4 Using Winsock in Server Applications

To complete the client example, we will develop an ActiveX-based timeserver application. One issue with the Winsock control is that it can only handle a single connection at a time. This could potentially be a problem with TCP-based server applications because a connection is needed for each instance. The solution is to have multiple instances of the Winsock control to handle multiple connections. This is pretty much in line with the LabVIEW TCP API. We will need to support the Connection Request event. Connection Request returns a connection ID, and any instance of the Winsock control can accept the connection. When a connection
request occurs, we generate an instance of the control and call its Accept method with the ID of the requesting connection. An array will be used to track all the instances of the control. When one instance of Winsock is loaded into memory, the system will not need to reload the ActiveX control from the hard drive. This is true of all ActiveX controls, but it is not very common to use multiple instances of ActiveX controls and generate them on the fly. A Winsock server is an example of an application that needs to generate copies of itself on the fly.

8.6.5 Using Winsock for Multiple-Connection Servers

When you have multiple copies of the Winsock control running, events will come back from each of the control copies. There are two alternatives to handle event cutbacks. One is to have a fixed number of loops, one per Winsock control. This is fairly easy to implement, but requires a fair amount of space on the block diagram and limits the number of Winsock controls you can create to the number of handling loops that you create. The second technique is to use a polling loop (ack!) and cycle through an array of event refnums with short timeout values for the Wait On event. The second technique allows more flexibility, but does not allow you to suspend the code diagrams as effectively because of the polling loops.

To illustrate using two Winsock controls without polling loops, examine Figure 8.14. This type of configuration has two Winsock controls. The first control is used to handle incoming connections, and the second to handle the requests. If a second connection request is received it will be denied because the worker control and listener control are busy.

To use the short timeout polling loop for multiple connections, check out Figure 8.15. The short timeout allows each connection to have the opportunity to check for new event occurrences. Internally, a Handler function that has access to the individual Winsock control refnums and other state information can be used to process individual messages. Depending on the application, the handlers can be simple to fairly elaborate. The main Winsock control is used to listen to a known port, and passes off connection requests to new Winsock controls. The “child” Winsock controls can be created and reused in a pool, or can be created and destroyed on the fly. The advantage of a pool of Winsock controls is that your code does not spend any time creating or destroying instances, but it does have a strict limit on how many connections can exist. Creating and destroying controls on the fly allows more flexibility in the amount of memory used and the number of connections that can be handled.

8.7 Microsoft System Information Control

It has always been possible through API calls to identify which version of the Windows operating system your code was executing. An easy interface to gather this information is presented by the Sysinfo control. This control has a variety of information regarding the operating system, power status, and work area. The Sysinfo control does not have any methods, but does have a list of interesting properties and events. The events are not necessary to use the control; all properties can be accessed without needing to respond to events.
Designing this driver collection will require four VIs for the properties. We will have one VI to open the control, one to handle power status, one to give operating system information, and the last to give work area information. No Close VI will
be built. There is no obvious need to wrap the automation close in a VI; that will just add extra overhead.

NASA-certified rocket scientists would not be needed to verify that the code diagram in Figure 8.16 simply creates a connection to the control. We will not be asking for any properties from the control at this time. Opening ActiveX controls should be done as soon as possible when an application starts up. This allows us to open the control at the beginning of a run and query information from the control when it becomes necessary.

The Power Information VI code diagram in Figure 8.17 is nearly as simple as the Open statement. The values that can be returned are AC Status, Battery Full Time, Battery Status, Battery Percentage, and Battery Life. All return values are in integers, and we will now explain why and what they mean. AC Status has three possible values: 0, 1, and 255. Zero means that AC power is not applied to the machine, 1 means that AC power is being supplied, and 255 means the operating system cannot determine if power circuitry is active. Battery Full Time is actually the number of seconds of life the battery supports when fully charged; a value of –1 will be returned if the operating system cannot determine the lifetime the battery is capable of. Battery Percentage is the percentage of power remaining in the battery. A return value of 100 or less is the percentage of power remaining. A value of 255 or –1 means that the system cannot determine the amount of battery life left. Battery Life returns the number of seconds remaining of useful power from the battery. A value of –1 indicates that the system cannot determine this information.

This information is only useful for laptop or battery-powered computers. The average desktop computer has two power states: AC on or AC off. If the desktop is hooked up to an uninterruptible power supply (UPS), we will not be able to get to the information from the Sysinfo control. Some UPS devices have interfaces to the computers they support, and a different driver would be needed for that type of monitoring. Consult the documentation with your UPS to determine if a driver is possible.

The next set of properties, the operating system information, may be useful when applications you are running are on different variants of Windows. Currently, there are more variations of Windows than most people realize. Consider that there are three versions of Windows 95 that could be run: the original Windows 95, Windows 95 with Service Pack 1 applied, and Windows 95 Operating System Release 2 (OSR 2). Windows 98 has two releases at the time of this writing, and there are a total of five service packs available for Windows NT 4.0. If you are doing hard-core programming involving ActiveX, OLE, or custom DLLs, the operating system information may be useful for troubleshooting an application. A second option is to dynamically change your code based on the operating system information.

FIGURE 8.16
Operating system properties available are the version, build number, and platform. “Platform” indicates which Win32 system is running on the machine: 0 represents Win32, 1 represents Windows 95, and 2 indicates Windows NT. The “Build Number” indicates which particular compile number of the operating system you are running. Actually, this property will not be tremendously useful because the build number is not likely to change. Service packs are applied to original operating systems to fix bugs. The version number also indicates which version number of Windows you are currently running. Windows 95 is considered Windows version 4.0 (it followed Windows 3.0 and 3.1).

The work area properties are useful if you programmatically resize the VI. This information gives the total pixel area available on the screen considering the task bar. This way, panels can be scaled and not occupy the area reserved for the task bar.

LabVIEW 5.1 users can take advantage of the events this control supports. Many of the commands are not going to be of use to most programmers. The operating system will trigger events when a number of things are detected. One good example is that when a PCMCIA card is removed from a laptop, the operating system will detect this (although not as quickly as many programmers would want). Once the system has detected the removal of a PCMCIA card, and you have an event queue for this control, you will be informed by the operating system that a PCMCIA card has been removed. This type of functionality will be useful for developers who need to support portable computer designs. PCMCIA card removal notification would allow an application to stop GPIB or DAQ card-handling and would be useful for adding robustness to an application. For example, code that could be executed once this event happens would be to halt reads, writes, or commands that route to the PCMCIA card until a PCMCIA Card Inserted event is generated. The event handling for this control is left up to the reader to look up. The system control should be supplied with most systems.

8.8 MICROSOFT MAPI SERVICES

Microsoft MAPI is Messaging Application Programming Interface. This set of controls is used to enable mail aware-applications. Two controls are used: the message and connection controls. Both must be used to develop the messaging driver. This driver is useful for e-mailing results of code execution. Some paging services also route text e-mail messages to pagers. E-mail code is significantly easier to
develop than code to control a modem and dial a pager. MAPI does not use events, therefore this driver is usable in LabVIEW 5.0.

The MAPI interface requires that Microsoft Exchange or Outlook be installed on your system. Configuration information, such as the mail server, needs to be configured prior to using the controls. Configuration information can be located in the Control Panels folder under “Mail.” Documentation for this control suggests MAPI configuration is temperamental—errors that are encountered could be configuration issues.

The MAPI Session control is used to log into the messaging subsystem of Win32. Without a valid connection, the controls will not do anything useful. The purpose of the Session control is to give the application access to the server, and information such as the contents of the user’s inbox. First, we need to create the MAPI session, then log into the network. The MAPI Session Open VI presented below takes a user name and password as inputs. In addition to the user name and password properties, we will set the new session property to be “true,” indicating that this is a new session. A Logon VI property is set to “false.” This property determines if the MAPI session should prompt a user for the logon information. This driver is intended for automated mail generation for user feedback; we are not implementing user prompts in this example. Next is to call the Sign On method. This method causes the MAPI subsystem to attempt a login to the mail server. If authentication fails, an error will be generated and returned in the error cluster. If user dialog was enabled, a dialog box would pop up asking the user to reattempt authentication. Again, this driver is intended for automated mail generation; dialog boxes are not desirable. The last step in the Open VI is to get the Session ID property. This property will be returned as an output of the MAPI Session Open VI. It will be needed by the messages control. Session ID has a default value of zero and is used to provide a unique handle to the control. It is possible to have multiple messaging sessions open concurrently.

The Session Close VI performs one step in addition to closing the automation refnum. We need to sign off or risk leaving resources on the server hanging. Busy server administrators might not appreciate this. Finally, a third method supported by the MAPI Session control is the About box. We are figuring that the average user will be impressed by a mail-aware application, but will not be impressed enough to read the credits. Therefore, the About Box method is left as an exercise for the reader.

The supporting session driver is not completed yet. There are a handful of properties that programmers may find useful. Other properties that could be useful are the Download Mail and Action properties. The Download Mail property determines if the MAPI session should download mail from the server when it arrives. We are leaving this at the value configured in the Control Panel. Some automated applications will want to force this value to be “false.” If the session does not process the mail, it could become lost or otherwise not respond. To prevent a potential problem, a MAPI No Download VI should be written.

8.9 MAPI MESSAGES CONTROL

Assuming that a valid connection to a mail server is established, it is time to put the session to good use. The Message control allows an application to generate new mail,
process received mail, and parse mail currently in the mailbox. This is quite a bit of power for a control; we encourage caution when using this type of control with your personal Outlook folders. This does not mean we discourage using this control, but when parsing the Inbox folder care should be used with the Delete function.

We will focus this driver collection on sending e-mail. The intended purpose is to e-mail automated reports, notify users that problems have occurred, and possibly send automated responses. The Open Message VI will use one argument: the session ID. We output this information from the Session Open VI, and here is where it is needed. The Message control needs to be bound to a particular session, and this information is carried in the session ID. Figure 8.18 shows the code diagram for this VI. We will leave configuration information such as recipient address, subject, and text to additional VIs. This is being done to leave flexibility in the driver. The driver can be used for sending and reading e-mail.

Before we proceed, we will explain how the MAPI Messages control is intended to work. Once the Session control has authenticated itself to the server, the Session control can give access to the user’s inbox. Two buffers are created: Compose and Read buffers. The Read buffer is read-only; you cannot alter the contents of messages currently in the inbox. The Message Index property identifies which e-mail in the inbox is currently being pointed to. When Message Index is set to –1, the Compose buffer is being pointed to. When generating an e-mail message, the Message Index property must be set to –1. Not to worry, the Message Index property will be set to –1 when the Compose method is called.

When writing the Open VI it should be obvious that there are 34 properties for the Message control. Many of these will not be implemented in our driver since we are intending it for message generation only. In fact, the process of generating a message is fairly simple: call the Compose method. Once this is done, set the recipient address, subject, add text, and then call the Send method. The driver uses a recipient name in addition to the recipient address. Both properties must have valid data. An empty string fed into the Name property will generate an error. Up to this point we have not mentioned anything about how mail is being transported through the MAPI subsystem. It does not matter which mail transfer protocol is being used, it is handled by the MAPI subsystem. We do not need to know or care how IMAP, SMTP, or POP3 work. To this point we have done everything except invoke the Send method.

It is possible with the Message control to set attachments. We left the possibility that a report may be generated in another application such as Word or Excel. By
not invoking the Send method, we are leaving room for attaching these files to the
e-mail and sending it afterward. Figure 8.19 shows the Attach Document VI. It is a
simple matter of setting the Attachment Path Name property. Do not confuse the
Path Name property with the Name property. The Attachment Name property simply
allows the programmer to specify a name to appear in the body of the message for
the attachment. This parameter is optional, and support for it is left as an exercise
to the reader. The Send method is called in a separate VI. This is a simple VI that
invokes the Send method of the control; its code diagram does not warrant an
illustration.

Figure 8.20 shows the VIs in action to send a simple e-mail message. Bill Gates
is informed that the control works well, and is thanked for giving the control away
with the operating system.

This control does have a number of error codes that it can return that are not
defined in winerror.h (these are error codes specific to this control). They are well
documented in the compiled Help file that is associated with the control. Compiled
Help files have CHM extensions and require a viewer from Microsoft to examine.
Unfortunately, this viewer is installed as part of a Visual Basic, C++ or MSDN
installation.

A few notes about this control should be presented before we move on to the
next example. This control can be very useful, but there are a few potential problems
you need to be aware of before you integrate it into your applications. First, when
the Send method is called, the message is placed in the outbox of your mail
application. This does not guarantee when the message is going to be routed to the
mail server. If Outlook or Exchange is currently running, the application will begin
routing the message immediately. When the mail application is not running, and
nothing is waking up the application, outbound mail will simply gather in the outbox.
A simple workaround is to start Outlook either manually or programmatically.
Outlook has an OLE interface that will start the application if you open a refnum
to it. This can be done at the start of an application to guarantee that your mail client
is active before you try to send mail.
8.10 MICROSOFT STATUS BAR CONTROL

The Microsoft Status Bar control is used to provide familiar status bar information to users. This control can be used in test applications to show the current status of processes currently running. The Status Bar control allows programmers to define different panels to supply basic information to users during execution. Many of the features of this control are useful for user interface designs, including concepts such as tool tips. When a user holds a mouse over an individual panel, a yellow text box will give basic information as to what the panel is describing. This type of information makes it easier for users to interface with applications. Most hardcore programmers consider this type of programming to be “fluff,” but user interfaces are the only aspect of an application the end users work with. Giving users the ability to see and understand more about what the application is doing is a relevant topic in any design. This control is being presented to lead us into the next control, the Tree View control. The Tree View control uses more complicated nesting and relationships between data members, so the Status Bar is a beginning point that leads into more complex controls.

The Status Bar is yet another user interface element, and its strong point is the standard Windows look with little programming needed. Selecting the properties of the control can configure many of the details on the bar’s appearance. Figure 8.21 shows the menu selection leading us to the status bar’s properties. We can select general properties to configure basic appearance information such as two- or three-dimensional separators for the control.

The Panels tab allows us to configure the panels that we wish to display to the users. Panels may contain text, tool tips, and images. Figure 8.22 shows the Tabs Configuration window for this control. Each panel is flexible in the sense that individual panels can be enabled and disabled without impacting the rest of the control. Panels can be beveled, which causes them to appear to “stick out” from the rest of the control, or inserted, which makes them appear to be “pushed in” relative to the other panels. Text can be centered, or right-, or left-justified for appearances. All of these properties can be configured for startup and altered during run-time.

In order to work with the control at run-time, we will require some kind of driver to interface to the control. The Status Bar control has a few methods and properties. The only property of the status bar we are interested in is the Panels property. This property returns an ActiveX refnum that give us access to the complete list of panels of the control. The Panels property is accessed like an ActiveX control, and Panels itself has properties and methods we can work with.

The Panels object has a single property: Count. Count indicates that it is a read and write property. You can set the count to any number you desire; the control will simply ignore you. This is an error in the interface in terms of the interface description: Count should be an “out” property, meaning that you can only read the value. Count returns the number of panels the bar is currently holding.

Methods for the Panels object include Add, Clear, Control Default, Control Default(1), Item, Item(1), and Remove. The Add method adds a new panel for you to place information. Clear removes all of the panels that are currently defined. The Control default has two different versions. The (1) that appears after the name
indicates that there are two sets of inputs that can be given to the method. ActiveX does not allow methods to have the same name and different arguments to the function. The function name and argument list is referred to as the “signature” of the method. ActiveX only allows the function name to have a single signature. The (1) makes the name different, so we are not violating the “one function, one signature” rule. The Control default gives access to the default panel of the control. We will not be using this method in our driver. The Remove method takes a variant argument similar to the Item method. This method removes the specified method from the status bar.
Item and Item(1) allow us to modify a single panel item. Both methods return a refnum to the specific panel. The Item method takes a variant argument that indicates which particular panel we want to work with. Our driver will assume you are referencing individual panels by numbers. The Item(1) method takes an additional parameter, a refnum to the replacement panel. For the most part, we will not be building new items to insert into the control, and we will not support the Item(1) method in our driver. Item returns yet another refnum pointing to the individual panel object for us to manipulate.

The Item object gives us access to the properties and methods for individual panels in the status bar. Items have no methods and 13 individual properties. We will be using most of these properties, or at least make them programmable in the driver. The Alignment property is an enumerated type that allows us to select Centered, or Right/Left Justified. The Autosize property configures the panel to automatically adjust its size to accommodate what is currently displayed. Bevel gives us the ability to have the panel “stick out” or to look “pushed in” relative to the other panels. Enabled allows us to make specific panels disabled while the rest of the control remains enabled. Index and Key allow us to set the index number for the panel or a text “key” which identifies the panel. We can use the key in place of the index number to reference the panel. This is desirable because, programmatically, it is easier to work with descriptive strings than integers. The Picture properties allow you to hand a picture reference to the control to change the currently displayed image. We will not be using the Tag items in this driver.

The driver itself is going to consist of a number of VIs to set the individual tasks. It is not likely that programmers will want to set each property of a panel every time they access it. Therefore, the driver collection should be structured with each property to a VI in order to maximize flexibility.

One design decision we need to reach before writing the driver is how we access the individual items in the control. We will generally be dealing with the item property of the Panels item of the control. There are a number of steps in the control, and we may decide that we are too lazy to continuously reference the control to reference the panels to reference the item. It makes sense to write a wrapper VI to encapsulate the traversal of the ActiveX objects. This type of wrapper will save us a bit of coding time, but an extra VI will create some performance hits on execution time. What we can do is write a VI that takes the status bar refnum and the item and traverses the objects, returning a refnum to the specific key. This VI can be run with subroutine priority to eliminate a lot of the VI overhead that is incurred when we use subVIs. We will use the variant argument as the front panel control so programmers will be free to work with either the index or a key name. The code diagram for this VI appears in Figure 8.23.

The VI to set the Tool Tip text for an individual panel item is shown in Figure 8.24. We call our get Item VI to return the refnum to the specific panel for us. This VI will be used in all of our driver VIs to obtain a reference to the desired panel item. We will not show the rest of the drivers because there is little additional information to learn from them, but they do appear on the companion CD to this book.
8.11 MICROSOFT TREE VIEW CONTROL

Assuming that you have worked with Windows Explorer, you are familiar with the basics of what the Tree View control does. Tree View is a mechanism to present data to users in an orderly, nested fashion. Information is stored into the control in the form of nodes. A node is simply a text field and optional bitmap. The Tree View control is capable of displaying images in addition to simple text fields. Potential applications of this control to LabVIEW developers are displaying configuration or test result information in a nested format. This control is part of the Windows common controls, and is arguably one of the most complex controls in this set.

Before we begin going into the details of how the Tree View control works, we need to get some background information on how the tree data type works. The Tree control stores data in a similar fashion to the tree data structure. As a data structure, trees store data in the form of leaves and branches. Each data point will be a leaf, a branch, or the root. Data items that point to yet other data items will be branches. The root is the first data element in the tree and is both the root and a branch. We will not display any illustrations of what a tree looks like; the easiest way to explain the data type is to instruct you to use Windows Explorer. This is a Tree View-based application and demonstrates well what the Tree View control looks like. At the root, you would have Desktop. All other elements in Explorer descend from the Desktop item. Desktop, in turn, has children. In the case of the computer on which I am currently typing, the desktop has six children. The children are My Computer, Network Neighborhood, Recycle Bin, My Briefcase and a folder titled, “LabVIEW Advanced Programming.” Each of these children, in turn, are branches and contain other data items which may or may not point to children of their own. This is how a tree data structure is set up. A special case of the tree structure has two children for each branch. This is known as a “binary tree.”

Each element in a tree contains some type of data and references to its parent and children objects. Data that can be stored in trees is not defined; this is an abstract data structure. Trees can be used to store primitive types such as integers, or complex
data types such as dispatch interfaces to ActiveX controls. In fact, the Tree View control stores dispatch interfaces to ActiveX controls. The Tree View control uses the dispatch interfaces to store references to its children and parent. Data stored in the Tree View control is a descriptive string and a reference to a bitmap.

Now that we have an idea how a tree structure works, we may begin exploring the operation of this control. A Tree View control needs to be inserted in an ActiveX container on a VI’s front panel. This control has short list of methods that we will not be using ourselves. Most of the methods of this control are related to OLE dragging and dropping. We do not intend to perform any of this type of programming, and do not need to implement driver VIs to support it.

The property listing for this control has a number of configuration items for the control itself, such as display options and path separator symbols. The Windows Explorer uses the plus and minus signs as path separator symbols. We will use them by default, but we have the ability to change them to whatever we desire. The main property that we will need access to is the Nodes property. In the Tree View control, each of the leaves and branches has been titled a “node.” The nodes contain all information that we need, such as the data we are storing and the location of parent, children, and siblings (nodes at the same level as the current node). Nodes serve as gateways to access individual data elements similar to the Items property in the Status Bar control. This is a fairly common theme to the architecture of many of Microsoft’s controls. You have not seen this type of design for the last time in this chapter.

The Nodes property is itself an ActiveX refnum. This refnum has the methods Add, Clear, Control Default, Item, and Remove. This configuration, again, is very similar to the Status Bar control, which will make learning this control much easier for us. Item and Control Default have properties with the (1) following them because they have two different sets of arguments that go with them. We will “stick to our guns,” and not use the methods that have the (1) following them because we are not going to track around different refnums for each of the nodes in this control. It is possible for us to have several hundred or even thousand items, and we will use the control itself to store the information.

Most methods use a variant argument called “Key.” Key is used to perform a fast lookup for the individual node that we want. The command for Clear does not use the Key argument, it simply dumps all of the contained nodes, not just a particular one. The Add method requires a few additional arguments, most of which are optional. Add has the arguments Relative, Relationship, Key, Text, Image, and Selected Image. We only need to specify Text if we desire. The control will go ahead and, by default, stuff the element at the end of the current branch. Item allows us to gain access to individual data elements stored in the tree control. This method takes a variant argument called Index and returns an ActiveX reference to the item we are requesting.

The Item object has only two methods that are related to OLE: dragging and dropping. We will not be using this method in this driver. There are a number of methods, however, that give us complete control of navigating the tree. We can identify the siblings of this item. Properties for first and last sibling give us the top and bottom element of the branch we are currently on. The next property gives us
a reference to the sibling that is below the current item in the list. This allows us to not track the keys for the individual elements in the tree and still work our way around it. The driver VI for this will be a single VI that has an enumerated type to allow a programmer to select which relationship is desired and return a refnum to that object. The code diagram is shown in Figure 8.25.

Individual elements can have text strings to identify their contents in addition to bitmap references. Users tend to appreciate graphical displays, and the bitmaps will make it easier for them to navigate the tree in search of the information they are interested in. For example, Windows Explorer uses bitmaps of folders and files to differentiate between listed items. Inserting images into the Tree View control is a bit of work. We need an associated Image View control that has the image, and we will pass an ActiveX refnum into the control. The Image View control is structured with the same item properties and methods as the Tree View and Status Bar Controls, and will not be covered in this book.

Now that we have mentioned how this control works, what properties and methods it has, and skipped out on the Image List control, we should begin to write and use the driver. We started this example by stating that this is only a user interface element. That is not really the case. You can use the Automation Open function to create the element, not display it and systematically store all configuration information into it. Functionally, it will operate very similar to the Dictionary control we presented earlier. Flattening any configuration information into a string will allow you to use this hive structure to store information. Unlike the Dictionary control, the Tree View control will allow you to have a directory-like structure to the information. One of the biggest problems we have seen with large-scale applications is having a good design for basic information storage. This control offers an interesting solution to this problem.

8.12 MICROSOFT AGENT

Microsoft Agent is a free ActiveX control that we can use to generate animated characters that can speak and decode audio input. These characters provide another element of a user interface design and behave somewhat similar to the Help characters in Microsoft Office. Unlike the Office characters, Agent’s characters do not reside in their own window; they are free-floating and may be set up anywhere on the screen. In addition, Microsoft Agent characters can be used by any application.
This type of user interface element is not something we expect to appear in requirement documents, but the control itself is a great example of COM programming.

This section will provide an overview of how this control works, and develop a driver collection to utilize it. All of the details on this control cannot be presented; enough information exists that a separate book can be written on this control. In fact, a book has been written for programming Microsoft Agent. Microsoft Press published *Programming for Microsoft Agent*, and additional information is available on Microsoft’s Web site.

Microsoft Agent is a service, meaning that it uses an out-of-process server and a control to reference the server itself. The server is responsible for controlling and managing the characters for different applications. This makes it possible to control Agent as a DCOM object. The characters can be made to appear on different machines to provide up-to-the-second status information.

Before we begin the design of the Microsoft Agent driver, we need to examine the design of Microsoft Agent itself. Microsoft Agent uses aggregated controls, meaning there are multiple controls for us to work with. The Agent control is a base starting point to interface to the server object, but most of the operations we will perform will be handled through ActiveX interfaces created by the Agent control.

Each of the aggregated controls represents a subset of the functionality of Microsoft Agent. Programming is simplified because each of the methods and properties are contained in appropriately-named controls. This design is similar to SCPI instrument command sets, where commands are logically grouped into a hierarchy of commands. Agent happens to follow the COM design methodology, and each of the branches in the hierarchy happens to be a COM object.

We will first present the embedded controls with their methods and properties. Events will be covered after the main driver has been developed. This will allow LabVIEW 5.0 users to benefit from the driver, and allow LabVIEW 5.1 or later programmers to select which events they are interested in supporting.

### 8.12.1 Request Objects — First Tier

The first object that is of use to most programmers, but not as useful to LabVIEW programmers, is the Request object. The Request object is analogous to the error cluster in LabVIEW. All Agent commands return a Request object to indicate the status of the operation. This object is not as useful for LabVIEW programmers because each method and property invocation returns an error cluster, which contains similar information as the Request object. Languages such as Visual Basic do not have built-in error cluster-type support, and objects such as Request are trying to make up for that limitation. None of the elements of our driver will use the Request object. Each time we are passed a Request object we will issue an ActiveX Close on it.

The use of ActiveX Close is important when it comes to dealing with objects such as Request. If we do not close the object, LabVIEW will not call release on the Request object and a memory leak will result. All of the Request objects we do not close will reside in memory, and over long periods of time this will cause Windows to crash. This is not the fault of Windows, COM, ActiveX, or LabVIEW;
Programmers need to release objects when they are finished working with them so memory can be reclaimed by the system.

Programmers who are interested in using the Request object in their work will want to know that its properties include Status, Number, and Description properties. Status is an enumerated type that is similar to the status code in the stock LabVIEW error cluster. The Status property has four values instead of two, and includes information such as Successfully Completed, Failed, In Progress, and Request is Pending. The Number property is supposed to contain a long integer that contains an error code number. Description contains a text explanation of the current problem or status. As we mentioned, the Request object is an imitation of LabVIEW’s stock error cluster.

**8.12.2 Other First-Tier Controls**

The next four objects that are mentioned for completeness, but not used in our driver, are the Speech Input, Audio Output, Commands Window, and Property Sheet objects. Each of these serve purposes directly related to their names, but are not necessary to successfully program Microsoft Agent. Additional information on them can be located on Microsoft’s Web site, or in the book Programming for Microsoft Agent.

Properties that are available at the Agent control base are Connected, Name, and Suspended. The Connected property is a Boolean that we will need to inform the local control that we wish to establish a connection to the Agent server. We will set this property in our Agent Open VI. Name returns the current name assigned to the ActiveX control. This may be useful to Visual Basic programmers who can name their controls, but LabVIEW programmers work with wires. We will not make use of the name property. The Suspended property is also a Boolean and will indicate the current status of the Agent server. We will not make use of this property, but programmers should be aware of it as it would be useful for error-handling routines.

**8.12.3 The Characters Object**

The first tier object in the hierarchy that is of use to us is the Characters property. Again, this is an embedded ActiveX control that we need to access in order to get to the individual characters. The Characters object has three methods, and we will need to be concerned with all of them. The Character method returns a refnum to a Character control. The Character control is used to access individual characters, and is the final tier in the control. Most of our programming work will be done through the Character object. The Character method returns a refnum, and we need to keep this refnum.

The Load method is used to inform the Agent server to load a character file into memory. Agent server is an executable that controls the characters. For flexibility, the characters are kept in separate files and can be loaded and unloaded when necessary. We will need the Load method to inform the server to load the character we are interested in.

Unload is the last method of the Characters control. Agent server will provide support for multiple applications simultaneously. When we are finished with a
character, we should unload it to free up resources. Not unloading characters will keep the character in memory. This is not a mission-critical error, but it does tie up some of the system’s resources.

The last method is the Character method. This method simply returns a refnum to a character object in memory. An assumption is made that this character has already been loaded into memory. If this is not the case, an error will be returned. The refnum returned by this function is covered next.

8.12.4 The Character Control

We finally made it. This is the control that performs most of the methods that we need for individual character control. The Character control requires the name of the character as an argument to all properties and methods. This will become a consideration when we develop the driver set for this control. There are a number of embedded controls, and we need to track and keep references to the controls we will need.

The Character control supports methods for character actions. The Activate method allows the programmer to set a character into the active state. Active has an optional parameter, State, that allows the programmer to select which character is to be activated. We will assume the topmost character is to be activated in our driver.

Our driver will use the methods Speak, MoveTo, Play, Show, and Hide. These methods all cause the characters to take actions and add a new dimension to our application’s user interface. The method names are fairly self-explanatory, but we will briefly discuss the methods and their arguments. Speak takes a string argument and causes the character to display a dialog bubble containing the text. The control can also be configured to synthesize an audio output of the text. MoveTo requires a coordinate pair, in pixels, for the character to move towards. An animation for moving will be displayed as the character moves from its current position to the new coordinate pair. Play takes a string argument and causes the character to play one of its animations. Animations vary among characters, and you need to know which animations are supported by characters with which you choose to develop. Each of the Play arguments causes the character to perform an animation such as smile, sad, greet, and others. Hide and Show require no arguments and cause the character to be displayed or not displayed by the server.

There are a number of properties that can be set for Agent, but we are only going to need two of them. Visible returns a Boolean indicating whether or not the character is visible. This is a read-only property. If we want the character to be displayed, we should use the Show method. The other property we will make use of is the Sound Effects property. This property determines whether or not the Agent server will generate audio for the characters.

Now that we have identified all the relevant methods and properties we need to use Microsoft Agent, it is time to make design decisions as to how the driver set will be structured. Unlike the other controls presented in this chapter, Agent uses aggregated controls, and it is possible to have several different refnums to internal components of the control. This is not desirable; having programmers (including yourself) needing to drag around half a dozen different refnums to use the control
is far more work than necessary. We can pass around a refnum to the base control and the character name and rebuild the path of refnums back to the character as we use the control. Potentially, there are performance hits every time we go through the COM interfaces, but for user interfaces, performance is not an issue. Users still needs to drag around two pieces of information. We really only need one: the character name.

It stands to reason that the control should keep an ActiveX refnum as an internal global variable. This will allow programmers to keep names for multiple characters and run them through the same Agent control. This would be efficient on memory, because we only need to instantiate the control once. This also allows for different VIs that are running independently to use the same server connection, which is more efficient for the Agent server.

We will rebuild the paths back to aggregated controls for all calls, but as we decided before, performance is not a significant issue for this control. Most other controls do not support multiple connections; programmers need to make other instances of the control. Agent does not need multiple copies to run multiple characters. This example is going to show a different way of handling ActiveX controls. We will make this driver DCOM-enabled by allowing the programmer to supply a machine name for the control.

The first problem we encounter with using a global variable for a reference is multiple calls to our Open VI. We will use a technique called “reference counting.” Each time a user calls the Open VI, an integer count stored in the global variable will be incremented. If the number equals zero before we perform the increment, we will call ActiveX Open. If the number is nonzero, we will increment the reference count and not call ActiveX Open. This VI is shown for both cases in Figures 8.26 and 8.27. The Open VI returns only an error cluster; all other VIs will use the control through the global variable, and perhaps need to know the character name. When we need to open the control, we will set one property of the control. Connected will be set to “true.”

The Close VI works in a similar fashion, except we use the decrement operator. The decrement is performed before the comparison. If the comparison shows the number is not greater than zero, then we know we need to close the control. When the comparison after the decrement is greater than zero, we know that we need to store the new reference count value and not to close the control. C++ programmers typically use reference counting, and this technique has applications for LabVIEW programmers with ActiveX controls.

The first method we will write is the Load Character method. This method requires the name of the character to load and the load key. The load key we will use is a path to a file on a hard drive. Agent defines this as the load key, so the control can be used in ActiveX-enabled Web pages. The Load key can also be a URL pointing to a character file. Agent character files use an ACS extension. This VI simply builds the Load key from the character name and supplied path. We use the global variable to access the agent refnum, and then we access the character’s property. This property is used to gain access to the Load method. Once the Load method is performed, we close off the returned load value. Load’s return value is a Request object. We do not need this object, because any errors generated would be reported back in the error.
cluster. Folks like Visual Basic programmers who do not have an error cluster to work with need Request objects. This VI is shown in Figure 8.28.

It is important to note that we only closed off the Request object’s refnum. We are not finished with either the control or the character refnums; in fact, we have only begun to use both of these objects. Our next function to implement will be the Unload method.

The Unload VI will operate in a similar fashion to the Load VI. Again, we do not close off either the characters refnum or the agent control refnum; both of these objects are still necessary. The characters refnum is needed by other VIs that are making use of the Agent control, and the Agent control itself requires access to the Characters object. Dropping the reference count on this object could cause the buried control to be unloaded from memory, which would be a surprise to the main Agent control. The Unload VI is shown in Figure 8.29. Unlike the Load VI, Unload does not need the path to the character file, and does not return a Request object.
The last method of the Characters object is the Character method. We will not need a VI for this method; the driver will always handle it under the hood. Character is only needed when passing instructions to specific characters. Programmers will only need to pass the character name to our driver; the driver will handle the rest of the details.

Now that we have mentioned passing the character names in, it is time to start implementing the individual character controls for the driver. Characters need Show and Hide methods to enable them to be seen or hidden. The VI to perform this work is shown in Figure 8.30. Show and Hide will be implemented in a single VI to cut down on the total number of VIs that are necessary. Both commands return a Result object, and we promptly close this object.

Moving the Agent character around the screen will be easy to accomplish with the Move Character VI. We need three inputs from the user: the name of the character and the coordinates for the character to move to. You have probably noticed the cascading access to each of the elements. There is not much we can do about this; there is no diabolical plot by Microsoft to wreck the straight paths of our VIs. Visual Basic handles aggregated controls nicely. The LabVIEW method of tracking the error cluster clutters up our diagrams a bit. The Move To command in Visual Basic would look like this: result = Agent.Characters.Character("Taqi").MoveTo(100,100).

This is easy to read in Visual Basic, and the SCPI-like structure of the commands makes it very easy to understand what the command is doing with what portion of the Agent control.

The next command that can be implemented is the Speak command. This is going to be one of the more widely used commands for this driver. Speak causes the character to display a bubble containing the text, and, if configured, to generate synthesized audio for the text. Speak.vi is structured very similar to Move To but requires two strings for input. This VI is shown in Figure 8.31.

The last animation command we are going to support is Play. Play is a generic command that all characters support. Agent allows for flexibility by not defining what animations the characters must support. Different characters can have animations that are related to the specific character. For example, a dog character could have an animation named Bark, while Bark would not be of much use to a President Clinton character. The VI implementing this function is shown in Figure 8.32. Programmers will need to know which animations their characters support.

Now that we have an intact driver for Agent, we shall put it to work. The VI shown in Figure 8.33 shows a simple application that opens Agent, loads the Robby character, and has it speak. The 20-second pause makes sure that the character has time to speak before we unload the character and release the control. It is important
to let the character have time to complete its actions, or the server will cut out the character mid-sentence!

What makes the agent driver very easy to work with is that we encapsulated all details regarding ActiveX into wrapper VIs. The user’s application looks like any standard LabVIEW VI. It is not obvious that the COM subsystem is in use, or even that Agent is running on another machine. Good driver development for ActiveX controls can make some of ActiveX’s complexities a nonissue for programmers. When working with ActiveX, it is best to write a solid driver that makes the control look and feel like a standard LabVIEW VI.
This example may not be practical for many applications; dancing parrots may not be suitable for many production applications. We did manage to demonstrate programming techniques for advanced ActiveX controls. We saw a control that uses aggregation, and eliminated complexity from the driver for the benefit of the driver’s users. Reference counting was presented and used to minimize the number of instances we need of the control.

### 8.13 REGISTRY EDITING CONTROL

The Registry Editing control is perhaps the most dangerous control that will appear in this chapter. The Registry Editing control allows a programmer to get, edit, delete, and modify Registry keys. Only programmers who understand what the Registry is, and what not to do with it, should use this control. If you do not understand the Registry or are not comfortable with editing and modifying its contents, feel free to skip over this control. Folks who like to tinker with new ActiveX controls may want to backup the System Registry and some of their valuable data.

The Win32 Registry contains information necessary for booting the system and identifying the location of applications, components, and system services. Historically, most information needed to start an application was stored in an INI file. The System registry provides a new database storage mechanism to allow for a single file to handle major initialization of applications. Individual data items are stored in the form of keys. Each key takes on a particular value, such as a string or integer. ActiveX controls have several keys that are stored in the Registry. One of the parameters that each control must store is the location of the control itself. When LabVIEW uses an ActiveX Open, it has a name for a control, and that is about it. LabVIEW talks to the COM subsystem, which in turn contacts the System Registry to locate a component. Keys are grouped into directories called “hives.” There are five base hives: HKEY_CLASSES_ROOT, HKEY_USER, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE, HKEY_CURRENT_CONFIG, and HKEY_DYN_DATA. Application information is stored in the HKEY_CLASSES_ROOT hive, and generic user information is HKEY_USER. Information specific to the currently logged-in user is stored in HKEY_CURRENT_USER. The current configuration and dynamic data hives store information that we really, really do not want to look at or modify. HKEY_CLASSES_ROOT stores information about all applications and components in the system. If you examine the registry using the Registry Editor (Regedit) you will see the six base classes. Expanding the root key will show a hierarchy of folders that start with file extensions. Each file extension is listed in the Registry, and the system uses these keys each time you double-click on a file. When you double-click on a file with a VI extension, the system searches the root classes for the .vi key and determines that labview.exe is the file that should be used to open it. Needless to say, when storing information about your particular applications, HKEY_CLASSES_ROOT is the place to store it. It is strongly recommended that you not access other hives, it is possible to completely confuse a system and seriously confused systems need their operating systems reinstalled.
Many large-scale LabVIEW applications can take advantage of the Registry to store startup information. Information that is typically stored in the Registry would be information that is needed at application startup, and configuration information that does not change frequently. Examples of startup data that could be stored in a Registry is the name of the last log file used, which GPIB board the application is configured to use, and calibration factors if they do not change frequently during execution of an application. Data that should not be stored in the Registry are items that change frequently, such as a counter value for a loop. Accessing the Registry takes more time than standard file access, which is why it should be used at application startup and shutdown. Key values should be read once, stored locally in LabVIEW, and written back to the Registry when the application is exiting. During normal execution, Registry access would cause a performance hit.

The drivers for this control are amazingly simple. The Open Registry VI simply calls ActiveX Open and returns the handle to the control. It is perfectly safe to open this control; the refnum does not explicitly mean you are accessing the Registry. The Delete Key VI is one that should be rarely used, but it is included for completeness.

8.14 CONTROLLING MICROSOFT WORD

This example will give a brief overview of programming Microsoft Word through LabVIEW. Microsoft Office is designed to be programmable and extensible through Visual Basic for Applications (VBA). Complete coverage of all of Word’s automation abilities is well beyond the scope of this book, but we will try to give you a starting point for using Microsoft Word to add to your current applications. Controlling any of the components of Microsoft Office is not true ActiveX programming, it is actually OLE Automation.

Microsoft Word 97 and 2000 (Versions 8.0 and 9.0) have two main createable objects that LabVIEW can use. The first is the Application object and the second is a Document object. The Application object alone has 91 properties and 69 methods. The Document object has a whopping 125 properties and 60 methods. Obviously, we will not be covering all the properties and methods of these controls in this book. Also consider that a number of these properties and methods are aggregated controls of their own, which leaves the possibility of another couple hundred methods and properties for each control!

This dizzying array of methods, properties, and objects seems to make controlling Microsoft Word impossible. It really and truly is not. One of the Document properties is “Password.” Unless you are using password protection for your documents, you do not need to use this property. The Versions property is another almost-never used property; in fact, many Word users do not even know that Word has a version control system built in. What this proves to us is that we need to figure out which methods and properties we need to get up and running.

Both the Application and Document controls are available to perform different tasks. The Application object is intended to provide control for applications to the entire Word application, where the Document control is meant to allow a program the ability to work with a specific document. This control in particular tends to be
fussy about when components are closed off. ActiveX Close does not necessarily remove the DLL or EXE from memory it calls a function named Release. The Release function uses reference counting, similar to what we did in the Microsoft Agent driver, to determine if it is time to unload itself from memory. Microsoft Word itself will maintain reference for itself on all documents, but having multiple references open on some components seems to cause problems. Releasing references as soon as possible seems to be the best way to deal with this particular control.

To start with, we need to gain access to Microsoft Word. We do this by using ActiveX Open to create an instance of the Microsoft Word Application object. This will start Microsoft Word if it is not presently running (this is a strong clue that you are performing OLE automation and not ActiveX programming). We can make the application visible so we can see things happening as we go along. To do this, set the Visible property of Microsoft Word to “true.” Word just became visible for us (and everyone staring at our monitor) to see. Word sitting idle without any documents being processed is not a very interesting example, so we need to get a document created with data to display.

There are a number of ways we can go about getting a document available. Possible methods are to create one from any of the templates that Word has to work with, or we can open an existing text file or Rich Text Format (RTF) document. This brief example will assume that we want to create a new document based on the Normal template. The Normal template is the standard template that all Word documents derive from.

Since we have already opened a communications link with Word, we need to gain access to its document collection. The documents commands for Word are located in the Documents property. Like Microsoft Agent, Word uses aggregated controls to encapsulate various elements of the application’s functionality. Documents is analogous to the Characters property of the Agent control. Major document commands are available through this control, including the ability to select which of the active documents we are interested in working with. Methods available from the Documents object are Open, Close, Save, Add, and Item. Open, Close, and Save work with existing documents and Close and Save work with existing documents that are currently loaded into Word. The Item method takes a variant argument and is used to make one of the documents currently open the active document.

Our example will focus on creating new documents, so we will invoke the Add method to create a new document. Optional arguments to Add are Template and New Template. The Template argument would contain the name of a document template we would want to use as the basis for the file we just created. New Template indicates that the file we are creating is to be a document template and have a file extension of “DOT.” We will not be using either of these arguments in this example. The Add method returns an ActiveX refnum for a Document object. This refnum is used to control the individual document.

To save the document, we invoke the Save As method of the document refnum. The only required argument to this method is the file name to save the document as. There are a total of 11 arguments this function can take, and we have only made use of the File Name property. Other methods are there to allow programmers complete control of the application. The operations we have just described are shown in the code sample presented in Figure 8.34.
Thus far, we have managed to get control of Word, create a document, and save the document. We have yet to address the topic of putting text into the document to make it worth saving. Considering that the primary purpose of Word is to be a word processor, you would be amazed at how challenging it is to put text into a document. One issue that makes inserting text into Word documents difficult is the complexity of Word documents themselves. You are allowed to have images, formatting options, word art, and embedded documents from other applications—how do you specify where to put the text? Navigating through paragraphs is a trivial task when you are working with the application yourself, but programming the application is another story.

To insert text into the document we need the Range object. The range object is analogous to the mouse for applications. When users determine that they want to insert text at a particular location, they put the mouse at the location and click. This is much more difficult for a program to do. Selecting the Range method with the Invoke node produces a new reference. Once you have a reference to the range object, you can access its properties. To insert text in a particular location, you would select the Start property. This indicates where to start the insertion. Now that you have programmatically clicked your mouse on the desired location, you need to insert your text. By using the Text property for the Range object, you can enter a string containing the text you want to insert. If you will be making multiple insertions of text, you can use the End property. This property provides the index to the end of the text insertion. The index read here could be used to start the next text insertion.

Let’s assume that for some reason you need to perform a series of tests, take the results, and e-mail them to a group of users. We could use the MAPI control that we presented earlier, the SMTP driver we developed in chapter 3, the Internet toolkit, or Microsoft Word. Insert the test results into a Word document and then attach a routing slip to it. This will allow us to send the other users a Word document containing the test results, and we would only need to use one ActiveX control to perform the task. Using the Routing Slip property gives us access to the document’s routing recipient list.

If there is anything the preceding snippet shows us it is that anything we can do with Word on our own, we can do through the OLE interface. Typically, this is how application automation is intended to work. The VIs used to control Word appear on the companion CD.
8.15 MICROSOFT ACCESS CONTROL

This section will cover an example using ActiveX automation to write to a Microsoft Access database. Microsoft Access 8.0 Object Library Version 8.0 is used in this example. Access has many properties and methods in its hierarchy of services and can be complicated to use. This example will simply open an existing database, insert records into a table, and close the database. If you wish to perform additional operations with Access, you should find other reference material on controlling Access through automation. Because of its complexity, you can spend hours trying to accomplish a simple task.

Figure 8.35 displays the Upper Level VI, Insert Records into Access.vi, that is used to insert records into the Access database. The block diagram makes calls to the following VIs: Open Database.vi, Insert Records.vi, and Close Database.vi. The code for this task was broken up logically into these subVIs. The required inputs include Database Name, Table Name, and Data Cluster In. The Data Cluster is simply a cluster containing one control for each field in the database. The front panel with this data cluster is displayed in Figure 8.36. Since each control in the cluster is an array, this VI can be used after collection of each data point or after accumulation of all data to be shipped to the database. It is more appropriate for use after all data is accumulated to maintain efficiency since the references to Access are opened and closed each time this VI is called. Since this VI can be used if you have an existing database, it assumes you are familiar with Microsoft Access. Table 8.1 defines the steps you can use to create a new database and table with the fields used in this example.

After the database and corresponding table have been created, you can use the Open Database.vi to open a reference. The block diagram for this VI is shown in Figure 8.38. Microsoft Access 8.0 Object Library Version 8.0 was selected from the Object Type Library list from the dialog window for selecting the ActiveX class. There is only one createable object in this type library, Application (Application.Access.8). This indicates that this object is at the top of the hierarchy of services for the Access ActiveX server.

The Invoke node is then used to call the procedure to open the database. OpenCurrentDatabase is the name of the method selected from the long list of methods available. This method opens an existing database as the current database. The file path is the only required input. A database that resides on a remote machine can be specified if desired. The Exclusive input can be left unwired and defaults to “false” to open the database in shared mode.

The next method that is executed is CurrentDb. This function returns a database object which is essentially a reference to the database that is currently open. The reference, or pointer, can then be used to perform numerous operations on an open database. Both the Access._Application and CurrentDb refnums are passed out of this VI. The CurrentDb refnum is passed to the Insert Records.vi as a reference for the function. The block diagram of this VI is shown in Figure 8.39. This VI is responsible for generating and executing the SQL (Structured Query Language) string for inserting the records into the database.

The data cluster is sent to the database using the INSERT INTO command. This command is used to insert single records into the destination table. The syntax for
this command is as follows: INSERT INTO table name (field 1, field 2, ..) VALUES ('data 1', 'data 2', ..). The first half of the statement is generated outside of the For loop because it is constant. The data values are added to the statement inside the For loop. Each of the elements of the cluster is an array. The loop is auto-indexed to execute as many times as there are elements. The second half of the command is generated only as many times as needed to get the data across to the table.
Inside the For loop is an Invoke node with the Execute method selected. The Execute method performs an operation on the database pointer passed to it. Query is the only required input for this method. Any valid SQL Query action expression wired to the method will be executed. SQL expressions are used by various databases to perform actions. Examples of some SQL statements include INSERT INTO, SELECT, DELETE, and ORDER BY. The CurrentDb reference is then passed out of the VI.

Finally, the database and the refnums are closed in the Close Database.vi. The block diagram for this VI is displayed in Figure 8.40. The CloseCurrentDatabase method is executed to close the pointer to the current database that is open in Microsoft Access.

Other procedures are available if you want to write to databases through LabVIEW. One alternative is to utilize Microsoft DAO (Data Access Objects) in your
LabVIEW VIs. DAO objects are used to represent and access a remote database and perform the necessary actions on the database. These objects use COM and can be called through LabVIEW using Automation and the ActiveX container. ADOs (ActiveX Data Objects) are also available to perform similar actions with databases.

Another alternative available is the SQL Toolkit, an add-on package that can be purchased to perform database operations on various commercial databases. The toolkit simplifies the process of creating databases, inserting data, and performing queries. The SQL Toolkit comes with drivers for many popular databases. It requires you to configure your ODBC (Open Database Connectivity) settings by specifying a data source. This DSN (Data Source Name) is then used as a reference or pointer to the database. The DSN is a user-defined name that represents the database that will be used. This user-defined name is used in the SQL Toolkit VIs to perform the necessary actions programmatically. The toolkit comes with several template VIs that can be customized to get you up and running quickly with a database.

8.16 CONTROLLING LABVIEW FROM OTHER APPLICATIONS

It is time to discuss using other languages to control LabVIEW. From time to time it may be desirable to reuse good code you wrote in LabVIEW to expand the abilities
of an application that was written in another language. Control of LabVIEW VIs is through OLE automation. We will be using the same ActiveX interfaces that we have been working with for the last two chapters. As we mentioned in Chapter 7, OLE interfaces are a superset of ActiveX interfaces. Both technologies rely upon COM’s specification.

We will be controlling LabVIEW through Microsoft Word. Word 97 and Word 2000 both have Visual Basic for Applications (VBA) built into their cores. Users still working with Word 6.0 will not be able to perform any of these examples because Word 6.0 relies upon Word Basic, which does not support COM interfaces.

The first issue we learn about quickly is that Visual Basic and Visual Basic for Applications have no idea where LabVIEW’s type library is. By default, you will not be able to use Intellisense to parse through the commands available. This is easily fixed; we need to import a reference to LabVIEW, and then we can get into the Intellisense feature that makes VBA a great language to write scripts with. Start by bringing up Word 97 or 2000. Make the Visual Basic toolbar active by right-clicking in an empty space next to the Word menu. Select Visual Basic, and a floating toolbar will make an appearance. The first available button is the Visual Basic Editor button. Click this button to start up the VBA editor. The VBA editor is shown in Figure 8.41. When performing this type of Word programming, we STRONGLY recommend that you do not append macros to the Normal template. The Normal template is the standard template that all Word documents begin with. Adding a macro to this template is likely to set virus detection software off every time you create a document and hand it over to a co-worker. This template is monitored for security reasons; it has been vulnerable to script viruses in the past. Right-click on Document 1 and add a code module. Code modules are the basic building blocks of executable code in the world of Visual Basic. Visual Basic forms are roughly equivalent to LabVIEW’s VIs; both have a front panel (called a form in Visual Basic) and a code diagram (module in Visual Basic). This new module will be where we insert executable code to control LabVIEW.

Now that we have a module, we will enable Intellisense to locate LabVIEW’s objects. Under the tools menu, select References. Scroll down the list of objects until you locate the LabVIEW 5.0 or 5.1 type library (depends on which version of LabVIEW you are running). Click on this box, and hit OK. The reference box is shown in Figure 8.42.

LabVIEW exposes two main objects for other applications. The first is the Application object, which exposes methods and properties global to LabVIEW, and the Virtual Instrument object, which gives access to individual VIs. In order to have an object pointing to the LabVIEW application you need to create an object. Objects in VBA are created with the DIM statement, with the format DIM <name> as <object>. When you type “DIM view as...” you will get a floating menu of all the objects that VBA is currently aware of. Start by typing “L” and LabVIEW should appear. The word “LabVIEW” is simply a reference to the type library and not a meaningful object. When the word LabVIEW is highlighted in blue in the pull-down menu, hit the Tab key to enter this as your selection. You should now see DIM view as LabVIEW. Hitting the Period key will enable a second menu that you maneuver through. Select the word “Application” and we have created our first object of type
LabVIEW application. Here is a helpful hint that gave the authors fits while trying to make a simple VBA script work: just because you have created a variable of type LabVIEW application does not mean the variable contains a LabVIEW application! Surprise! You need to inform VBA to set the contents of the variable, and you need the CreateObject method to do this. The next line of code needs to read “Set view = CreateObject(“LabVIEW.Application.5”).” This will cause VBA to open LabVIEW and create the OLE link between the applications. Now we are in business to open a VI. Dimension a new variable of type LabVIEW.VirtualInstrument. This is, again, an empty variable that expects to receive a dispatch interface pointer to a real VI. We can do this with a simple Set command. The Set command needs to look like “Set
vi = view.GetViReference(“Test.VI”).” this command will instruct LabVIEW to load the VI with this name. The line of code we gave did not include a path to the VI. Your application will need to include the path. The GetViReference returns a dispatch interface pointer to the variable. The interface pointer is what VBA uses to learn what methods and properties the Virtual Instrument object exposes.

Now that we have our channel to LabVIEW and a VI loaded into memory, we can set front panel controls and run the application. LabVIEW does not provide a mechanism to supply the names and types of front panel controls that exist on a particular VI. You need to know this information in advance.

There are a couple of methods that we can use to read and set the controls for a front panel. We can set them individually using a Set Control method. Set Control requires two properties; a string containing the name of the control to set and a variant with the control value. This is where variants can be a pleasure to deal with. Since variants can contain any type, we can pass integers, strings, arrays of any dimension, and clusters as well. Did we say you can pass clusters out of LabVIEW code? Yes, we did. Unlike dealing with CINs and DLLs, the OLE interface treats clusters like arrays of variants. Popping up on the cluster and selecting Cluster Order will tell you the order in which the cluster items appear.

Higher-performance applications may take issue with all of the calls through the COM interface. To set all parameters and run the VI at the same time, the Call method can be invoked. Call takes two arguments; both are arrays of variants. The first array is the list of controls and indicators and the second array is the data values for controls and the location to store the indicators’ values when the call is done. Call is a synchronous method, meaning that when you execute a Call method, the function will not return until the VI has completed execution.

To call VIs in an asynchronous method, use Run, instead. The Run method assumes that you have already set the control values in advance and requires a single Boolean argument. This argument determines if the call will be synchronous or asynchronous. If you do not want this function call to hold up execution of the VBA script, set this value to “true.” The VI will start executing and your function call will return immediately.

LabVIEW exposes all kinds of nifty methods and properties for dealing with external applications. It is possible to show front panels and find out which subVIs a VI reference is going to call. You can also find out which VIs call your current VI reference as a subVI. Configuring the preferred execution subsystem is also possible if you are in a position to change the threading setup of LabVIEW. This is discussed in Chapter 9.

To find out more information on what methods and properties LabVIEW exposes, consult LabVIEW’s on-line reference. There is complete documentation there for interfacing LabVIEW to the rest of the COM-capable community.

8.17 UNDERSTANDING ACTIVEX ERROR CODES

ActiveX and the COM subsystem have their own set of error codes. Virtually every ActiveX function has a return type that LabVIEW programmers cannot see. In C and C++, functions can only have one valid return type. COM functions generally
use a defined integer called an HRESULT. The HRESULT is a 32-bit number that contains information about how well the function call went. Similar to LabVIEW error codes, there are possibilities for errors, warnings, informative messages, and no error values. The first two bits of an HRESULT contain the type of message. If the leading bit of the number is set, then an error condition is present. The lower 16 bits contain specific error information. In the event an ActiveX call goes south on you, you can get access to the HRESULT through the error cluster. The cluster will contain a positive error condition, the HRESULT will be the error code, and some descriptive information will be in the error description.

If you have access to Microsoft Visual C++, the error code can be located in winerror.h. Winerror.h lists the errors in a hexadecimal format. To locate the error in the file, you must convert the decimal error code to hex value and perform the search. The dispatch interface error (0x8000FFFF) is one of the more commonly returned error types, a “highly descriptive” error condition that is used when none of the other codes applies. Debugging ActiveX controls can be challenging because the error codes may be difficult for LabVIEW programmers to locate.

Some error codes are specific to the control. This will be made obvious when the string return value in the error cluster has information regarding where to find help. As a general troubleshooting rule, it helps when documentation for the control is available, and possibly the use of Visual Basic. LabVIEW has a custom implementation of the ActiveX specification and does not always behave the same as Visual Basic and Visual C++. We have seen cases in custom ActiveX controls where LabVIEW’s interface was more accepting of a control than Visual Basic or the ActiveX test container supplied with Visual C++.

When troubleshooting an ActiveX problem, the obvious issue is that it is impossible to see inside the control. The first step is obvious: check the documentation! Verify all inputs are within spec, and that all needed properties are set. A number of controls have requirements that certain properties be initialized before a Method call can be successful. For example, in the Winsock driver, a call to connect must be made before a string can be written. Obvious, yes, but one author forgot to do this when testing the driver. After several reloads of LabVIEW and rebooting the system, the error was identified.

One thing to understand when looking at error codes is that Automation Open usually does not fail. The reason for this is that this VI will only report problems if the OCX could not be loaded. An ActiveX control is simply a fancy DLL. The act of calling Automation Open loads this DLL into LabVIEW’s memory space (consult the multi-threading chapter for more information on memory space). Close usually will not generate errors either. The Automation Close does not unload the DLL from memory, which came as a surprise to the authors. ActiveX controls keep a reference count; the Automation Close VI calls the Release function, but does not eliminate the control from LabVIEW’s memory space. It is possible to get a control into an odd state, and exiting and restarting LabVIEW may resolve problems that only occur the second time you run a VI. Unfortunately, the only method we have found to force an ActiveX control out of LabVIEW’s memory space is to destroy LabVIEW’s memory space. The solution you need to look for is what properties/methods are putting the control into an odd state and prevent it from happening.
National Instruments had to supply its own implementation of the ActiveX specification. This is not a bad thing, but it does not always work identically to Microsoft’s implementation in Visual Basic. The second recommended step of escalation is to try equivalent code in Visual Basic. When writing the MAPI driver, we discovered that example code written in LabVIEW just did not work. We tried the equivalent code in Visual Basic and, surprise! It also did not work. Reapplying the first rule, we discovered that we were not using the control correctly.

In a different scenario, I was developing an ActiveX control in Visual C++. While hacking some properties out of the control, I did not pay attention to the modifications I made in the interface description file. The control compiled and registered correctly; LabVIEW could see all the controls and methods, but could not successfully set any properties. When I tested the control in Visual Basic and C++, only a few of the properties were available or the test application crashed. It appears that when National Instruments implemented the ActiveX standard it prioritized LabVIEW’s stability. LabVIEW would not crash when I programmed it to use a bad control, but Visual Basic and C++ test applications crashed. The moral to this story is directed to ActiveX control developers: do not develop ActiveX controls and test them only in LabVIEW. LabVIEW processes interface queries differently than Visual Basic’s and C++’s generated wrappers (.tlh files or class wizard generated files).

Programmers who do not “roll their own” controls can omit this paragraph. If you are using controls that you have programmed yourself, make sure the control supports a dual interface. Recall that LabVIEW only supports late binding. If the Idispatch interface is not implemented or behaves differently than the Iunknown interface, the control can work in Visual Basic, but will not work the same (if at all) in LabVIEW.

If a control does not work in either LabVIEW or Visual Basic, it is possible that the documentation for the control is missing a key piece of information. You will need to try various combinations of commands to see if any combinations work. This is not the greatest bit of programming advice, but at this point of debugging a control there are not many options left. Be aware of odd-looking error codes. If you get an error such as hex code 0x8000FFFF, varying the order of commands to the control is probably not going to help. This error code is the dispatch interface error and indicates that the control itself probably has a bug.

As a last resort, verify the control. If you cannot get a control to work properly in Visual Basic or LabVIEW, some sanity checks are called for. Have you recently upgraded the control? We experienced this problem when writing the Microsoft Agent driver. Apparently, Registry linkages were messed up when the control was upgraded from Version 1.5 to Version 2.0. Only the Automation Open VI would execute without generating an error. If possible, uninstall and reinstall the control, rebooting the machine between the uninstall and reinstall. It is possible that the Registry is not reflecting the control’s location, interface, or current version correctly. Registry conflicts can be extremely difficult to resolve. We will not be discussing how to hack Registry keys out of the system because it is inherently dangerous and recommended as a desperate last resort (just before reinstalling the operating system).
If none of the above methods seem to help, it is time to contact the vendor (all options appear to now be exhausted). The reason for recommending contacting the vendor last is some vendors (like Microsoft, for example) will charge “an arm and a leg” for direct support. If the problem is not with the component itself, vendors such as Microsoft will ask for support charges of as much as $99/hr of phone support.

8.18 CONTROLS THAT DO NOT WORK WELL WITH LABVIEW

This is a topic we will briefly cover so readers know a handful of controls that will not work with LabVIEW, and also why LabVIEW cannot support the controls. Top on the list of controls that LabVIEW programmers may not use are Container controls. Any control that allows for additional ActiveX controls to be placed inside of it is not useful to us. An example of this is the Tab control. The Tab control gives programmers the ability to make tabs and place components onto the corresponding pages of each tab. We do not have the ability to insert controls into the tab strip, therefore we cannot make use of it.

If you are hellbent on making a tab strip work in LabVIEW, you do have a few possible workarounds to use. The first is to lay out the tab strip with the appropriate tabs. Lay out all of the controls for the first tab on the front panel. Creating an Attribute node for each control will allow you to manually make the control visible and invisible. Each time a user selects an option on the tab, you will need to toggle all of the Attribute nodes. We did not say this was a pretty solution, but it can allow you to work with the control.

The second, and potentially easier, workaround is to generate a custom ActiveX control in Visual Basic. Lay out the tab strip and embed the ActiveX controls into the custom control’s tab strip. Very large numbers of elements will make for a huge property listing for the control, and you will need to write your own event code to handle clicks and tab changes. Again, this can be a significant amount of work, but it is possible to work with the tab control if you really need to.

Microsoft’s Cool Bar control is another control that we cannot directly work with; in fact, the control will identify that LabVIEW cannot support nesting controls and will not allow itself to be loaded. We do not have a workaround to present on this control, and users are going to have to wait for a future version of LabVIEW to work with it.

A handful of third-party controls may not appear to work correctly with LabVIEW. Troubleshooting these controls can be difficult. The easiest way to start is to try to get Visual Basic to work with the control. We are not suggesting that you need to spend $500 on Visual Basic if you do not have a copy of the application. You can try to get the control to work with an application that supports Visual Basic for Applications, such as any of the Microsoft Office applications. Assuming one of these applications is installed on your machine, try to get the control to work with VBA. If the control works, then we can assume that the control works (kind of).

The suspect control obviously does not completely support the ActiveX specification or it should work without issue in LabVIEW. A workaround that you can
use to try to get the control working in LabVIEW is to wrap the ActiveX call in a wrapper VI. Each parameter passed should be linked to a variant input on the front panel of the wrapping VI. Your code passes data to the variant inputs on the wrapper, and the variants will be passed to the control. Hopefully, the control will handle the variant inputs and we might just get the control working.

### 8.19 ADVANCED ACTIVEX DETAILS

This section is intended to provide additional information on ActiveX for programmers who are fairly familiar with this technology. LabVIEW does not handle interfaces in the same manner as Microsoft products such as Visual C++ and Visual Basic. Some of these differences are significant and we will mention them to make ActiveX development easier on programmers who support LabVIEW. Intensive instruction as to ActiveX control development is well beyond the scope of this book. Some information that applies strictly to LabVIEW is provided.

The three major techniques to develop ActiveX controls in Microsoft development tools are Visual Basic, Visual C++’s ATL library, and Visual C++’s MFC ActiveX Control Wizard. Each of the techniques has advantages and disadvantages.

Visual Basic is by far the easiest of the three tools in which to develop controls. Visual Basic controls will execute the slowest and have the largest footprint (code size). Visual Basic controls will only be OCX controls, meaning Visual Basic strictly supports ActiveX. Controls that do not have code size restrictions or strict execution speed requirements can be written in Visual Basic.

The ActiveX Template Library (ATL) is supported in Visual C++. This library is capable of writing the fastest and smallest controls. Learning to develop ATL COM objects can be quite a task. If a developer chooses to use custom-built components for a LabVIEW application, budgeting time in the development schedule is strongly recommended if ATL will be used as the control development tool. ATL is capable of developing simple COM objects that LabVIEW can use. LabVIEW documentation only lists support for ActiveX, but ATL’s simple objects can be used in LabVIEW. Simple objects are the smallest and simplest components in the ATL arsenal of component types. Simple objects are DLLs, but we have used them in LabVIEW. The interesting aspect of simple controls is that they will appear in LabVIEW’s list of available controls. They do not appear in Visual Basic’s list of available controls; they must be created using Visual Basic’s CreateObject command.

The MFC support for ActiveX controls will generate standalone servers or .ocx controls. When designing your own controls, MFC support may be desirable, but developing the object itself should be done with the ATL library. Going forward, ATL seems to be the weapon of choice that Microsoft is evolving for ActiveX control development. As we mentioned, it is possible to get the support of the MFC library built into an ATL project.

Components that will be used by LabVIEW must be built with support for dual interfaces. Single interface controls will not work well in LabVIEW. Visual C++ will support the dual interface by default in all ActiveX/COM projects. A word of caution is to not change the default if your control will be used in LabVIEW. We
have previously mentioned that LabVIEW only supports late binding. Controls that only have one interface will not completely support late binding.

ActiveX controls have two methods used to identify properties and methods: ID number and name. LabVIEW appears to address properties and methods by name, where Microsoft products use ID. This tidbit was discovered when one of the authors hacked a property out of a control he was working on. This property happened to have ID 3. LabVIEW could see all the properties in the control, but Visual Basic and C++ could only see the first two properties in the list. The Microsoft products were using ID numbers to identify properties and methods, but LabVIEW was using names. The lesson to learn here is that you need to exercise caution when hacking Interface Description Language (.idl) files.

It is possible to write components that accept LPDISPATCH arguments. LPDISPATCH is a Microsoft data type that means Dispatch Interface Pointer. LabVIEW can handle these types of arguments. When an LPDISPATCH argument appears in a method, it will have the coloring of an ActiveX refnum data type. All of the aggregated controls shown in this chapter use LPDISPATCH return types. You, as the programmer, will need to make it clear to users what data type is expected. The workaround for this type of issue is to have the control accept VARIANTs with LPDISPATCH contained inside. This will resolve the problem.