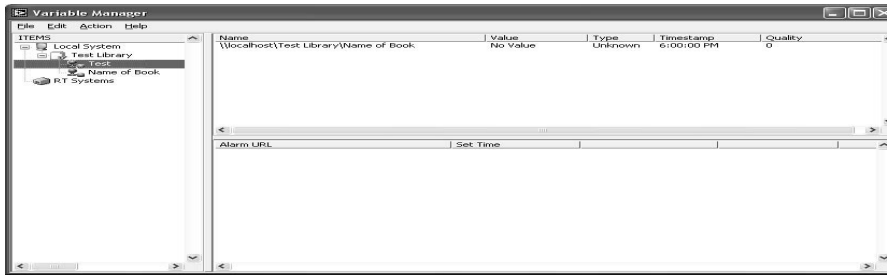


FIGURE 7.1

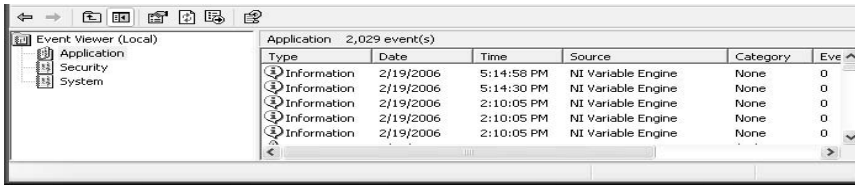


FIGURE 7.2

The watch list pane allows for viewing the status of individual variables. Specific elements that can be seen in the watch list are the name of the variable, the current value, its type, the time stamp of last value change, and the quality.

The variable manager allows for a user to add and configure variable libraries and variables without running LabVIEW. It also allows for libraries to be created from outside the context of a LabVIEW project.

7.2.1.2 Windows Event Viewer

The second tool for accessing the health of the engine is the event viewer. From the Windows Control panel, access administrative tools. The events viewer shows system recorded events for applications, security, and the system. Engine events are recorded in the application window. An example of the application counter is shown in Figure 7.2. The events recorded are fairly generic in nature, but will give an administrator knowledge of when the service was started or stopped.

7.2.1.3 Windows Performance Monitor

Another tool that can be used for monitoring the health or usage of the shared variable engine is the performance monitor. The performance monitor does not show directly items involving the shared variable engine but does provide some insights into how the engine and its host machine are performing. Section 7.4 discusses the networking aspects of shared variables, but for now one hint will be provided: the shared variable uses UDP as its transport protocol. The performance monitor is a Windows operating system tool that provides access to a large number of statistics about the health of the computer. Different elements of the operating system are broken up into "objects" such as the CPU, the hard drive, and networking statistics. One of value to us here

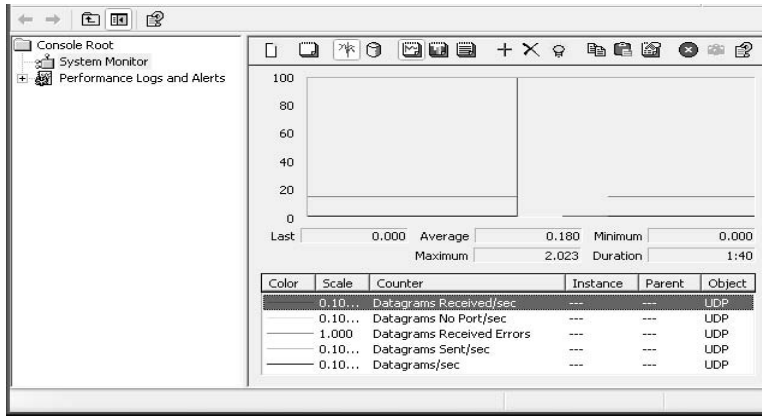


FIGURE 7.3

is the UDP performance. Figure 7.3 shows the performance monitor reporting on the volume of UDP traffic being processed by the computer. If a distributed application is experiencing problems, the performance tools can help identify what types of issues the networking may be experiencing. For example, if the error count is high, then there may be a problem with cabling or a switch in the network. As a reference, bit error rates on Ethernet are fairly low; on order of 1 in 1 million bits or less should be corrupted by a network. If you have hundreds of UDP messages per second arriving with errors, the problem probably exists in the network. The task manager's performance tab would supplement the performance counters. If the packet rates are large, compare this with the task manager's report of network utilization. A high utilization would suggest additional network capacity is needed.

One counter configuration that may be useful in determining the health of a networked application is shown in Figure 7.4. The performance monitor is tracking counters for the IP object, TCP object, and UDP object. Each object is displaying the Send and Receive counters. If there is suspicion that the host machine is very busy on the network — for example, it is hosting a web server for VI control and the shared variable engine — it can be expected to show the TCP and UDP Send/Receive counters running large numbers. It should not be surprising that the IP counters will run even larger numbers; in fact, it should be approximately equal to the sum of the UDP and TCP counters as IP is the underlying protocol of both TCP and UDP. The reason why the IP counter would only be approximately equal

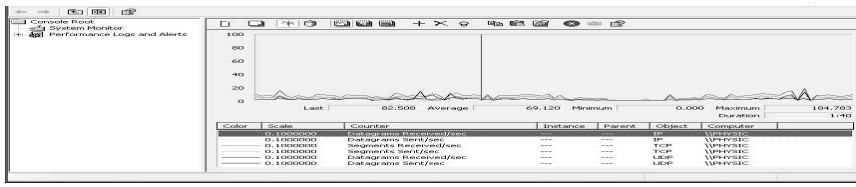


FIGURE 7.4

is because there is another commonly used protocol that also uses IP — ICMP. Ping messages and notification of message delivery failure are often sent through ICMP. It is possible for an administrator to monitor counts on multiple computers at once, so the counters shown above can be monitored for the shared variable engine and its clients.

7.2.1.4 Windows Task Manager

The Windows task manager is not really a tool to evaluate the health of the shared variable engine, but one tab on the task manager can yield some misleading information about the network utilization. When examining the Task Manager's Network Performance tab, it is reporting utilization of what the link is rated for, not what the network is capable of supporting. Most Ethernet links are 100 megabit or faster, but this does not mean the router the cable is plugged into is going to support this data rate. For example, the backbone of Rick's home network is a Nexland ISB SOHO router/firewall. The router supports 10/100 megabit operation on the links but is only capable of 6 megabit throughput internally. It would be very unlikely to see the Network Performance tab report over 6% utilization! Now this might sound clunky and old but consider the simple fact that it still has a higher throughput than many DSL and cable modem installations. When looking at the task manager's network utilization it is important to know what the other side of the Ethernet cable is capable of supporting. A remote client with a 128 kilobit DSL link may show 1.2% of its network being utilized, but that is all the DSL link has to offer. This may not be a factor in a typically modern engineering facility, but remote locations with high throughput may need to consider what their link to the system supports.

7.3 SHARED VARIABLE PROCESSES AND SERVICES

This section discusses the processes and services used in operating the shared variable. Several services are in use and provide different functional elements. Figure 7.5 shows some of the Windows services involved. Top of the services list is the Lookout Citadel server. The Citadel server is not dependent on the rest of the shared variable engine. It is installed with LabVIEW, but the DSC is needed to take full advantage of Citadel.

With the DSC module, additional capabilities available to distributed data include datalogging through the Citadel database, data alarms, data scaling, and initial values. In the event the shared variable engine is being used without the DSC

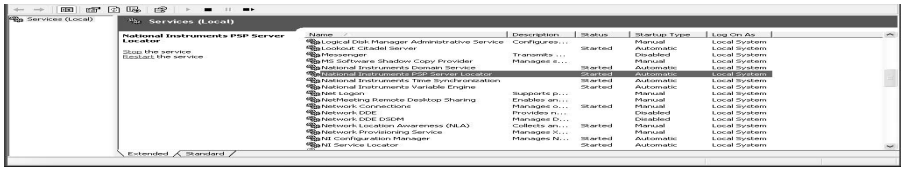


FIGURE 7.5

module, one of the first steps the application needs to do is set the shared variable's initial values on system startup.

Citadel is a streaming database and uses a feature called "deadbanding" for logging. If a value of a variable does not change, or does not change significantly, Citadel can be configured to not log the value update, so it is possible to have Citadel only log value changes of significance.

The DSC Module's alarm capabilities make this add-on desirable in many monitoring applications. The alarms have a significant level of configuration so the alarming feature can be tailored specifically to a scenario. Standard alarm configurations include values over high and low thresholds and a rate of change alarm. Rate of change alarms are useful in process control; for example, if a temperature sensor is reporting a mixture is heating too quickly or too slowly.

The National Instruments Domain Service is responsible for coordinating National Instruments security for the shared variable engine. This service is responsible for user logins and access to features of the shared variable such as who is allowed to stop library processes. Usage of this process is discussed in more detail in Section 7.7.

The National Instruments PSP Locater Service is responsible for coordinating the Publish-Subscribe-Publish Protocol transactions across system elements. It works with the NI security features that are part of the domain service. This service handles the detection of remote domains.

The National Instruments Time Synchronization Service is used primarily with Lookout or the DSC Module in LabVIEW. This module helps synchronization of time between remote machines; in general, there is no need to modify this service. In fact, it can be difficult to modify the time synchronization of a remote client without having either Lookout or the DSC Module installed. In cases where neither Lookout nor the DSC Module is installed, to configure this module to synchronize time to another machine, such as the one hosting the variable engine, use the following steps (note that this only works on Windows-based machines):

```
Start up a DOS window

enter: lktsrv -stop

enter: lktsvr -start <update interval> <IP Address>
```

Now this will work and cause the local time service to synchronize with the remote machine — but only until the computer is restarted; then this setting will be lost. To make a change like this "permanent," you can change the service Startup command in the Windows Services configuration. Go to the control panel and open up Administrative Tools. Open up the Services Applet and double click on the National Instruments Time Synchronization Service. A dialog window will open up and is shown in Figure 7.6. Stop the service, and add the second line shown previously into the start parameters box at the bottom of the window. Every time the computer is restarted, it will bring up the time service with the address specified. It is also possible to configure the service to synchronize time with multiple machines — separate the IP addresses using a space-delimited list.

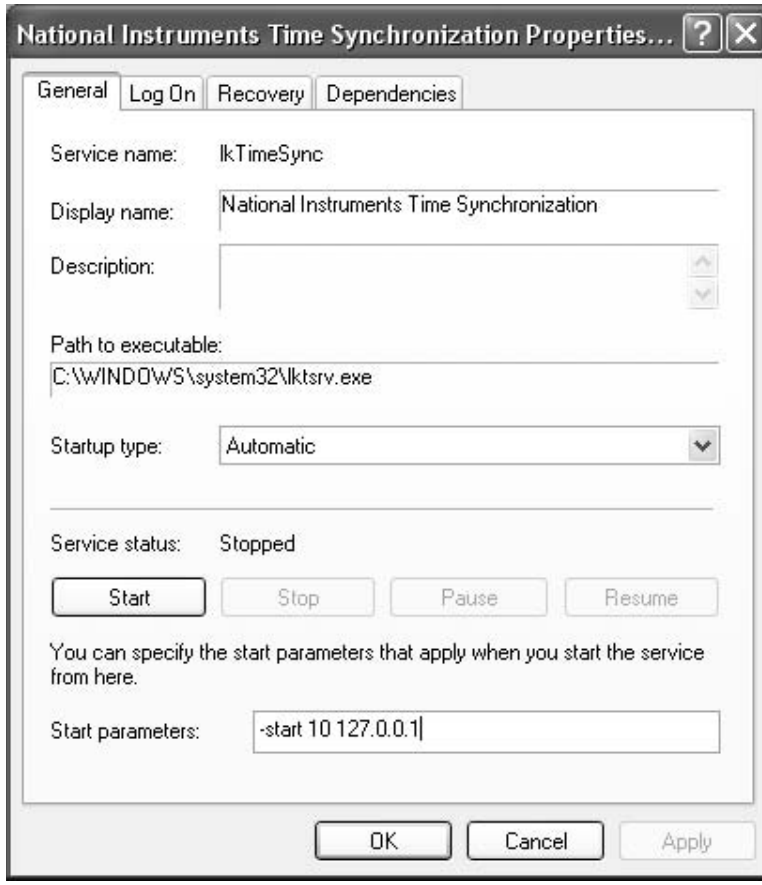


FIGURE 7.6

The National Instruments Shared Variable Engine is the last service to be mentioned, and it obviously implements the shared variable engine. This service is dependent on the NI Configuration Manager.

7.4 SHARED VARIABLE NETWORKING

Shared variables are transported across networks via a protocol called the NI Publish-Subscribe Protocol (PSP). The PSP protocol is proprietary in nature, so it is not possible to go into significant detail of message exchanges. There are several aspects of the protocol that can be discussed and we will begin with the underlying transport. The PSP protocol itself uses UDP/IP as the underlying transport protocol. UDP is not a reliable protocol; it is possible for PSP packets to be lost during transmission. The advantage to using UDP for the transport protocol is the lack of TCP-related overhead. TCP is reliable, but its windowing algorithm and associated acknowledgments can create what is called *network jitter*. When a TCP packet is lost, almost

every TCP stack in commercial use makes a fundamental assumption: the packet was lost due to network congestion. The TCP stack will then perform a delay and then attempt to retransmit the packet. The trouble with this mechanism is that the application sending the message is unable to control the retransmission delay. Retransmission delay is called *jitter*. Jitter is undesirable in shared variable applications because retransmissions end up delaying applications. If a front panel control is bound to a shared variable, the goal is to attempt to acquire a current value for the control. TCP-based communications could leave the application in a position of waiting around for the TCP windowing to push data up to the application.

Another disadvantage to using TCP is the requirement that the connection be open and maintained. The TCP socket needs to stay open, which requires the variable engine to maintain a port for the specific variable. A large, distributed application with dozens of shared variables would end up putting a load on the networking end of the shared variable engine. The choice of UDP as the transport protocol allows the engine to use a few ports across any number of remote applications. This is far more efficient from the perspective of the engine and allows the engine to concentrate on pushing data instead of maintaining TCP sockets.

As mentioned, UDP is not a reliable protocol because UDP itself does not provide notification that the message was received. The NI-PSP protocol implements acknowledgments and essentially resolves the unreliable aspect of UDP by implementing proprietary reliability functions on top of UDP. This may sound like considerable work, but, as stated above, TCP is not a desirable protocol when time can be a critical factor in network communications.

The engine and clients use sockets in the low 6000 range — specifically, ports 6000 to 6010 in addition to 2343. Basic network security requires the use of firewalls on every machine; these ports will need to be made available through the firewall. Applications using both shared variables and network communications need to avoid using these port ranges as they will be refused. The shared variable engine on the server side is a service, which means it will be up and running before the application; the application needs a user to launch it whereas the service comes on line before a user has logged in. This means the server will always win the race to request the ports from the operating system.

The format of data exchanged in the UDP packets is proprietary to National Instruments, meaning the exact message formats and field definitions are not public. What is known is that the PSP protocol uses uniform resource locators (URLs) to identify information about the shared variables available on a given engine. The engine will also forward out a lot of information about itself to a client during initial "conversation." This startup conversation includes which processes are responsible for different functions, including the process name, location, and version number down to the build number. Distributed timing is also a common issue in distributed applications; the engine notifies the client which process is creating time synchronization information.

The application in particular that is performing most of the server work is called "Tagger" and is located in Program Files\National Instruments\Shared\Tagger. In the event you are working on a machine using a software firewall such as Zone Alarm, you may receive mysterious requests about an application called Tagger

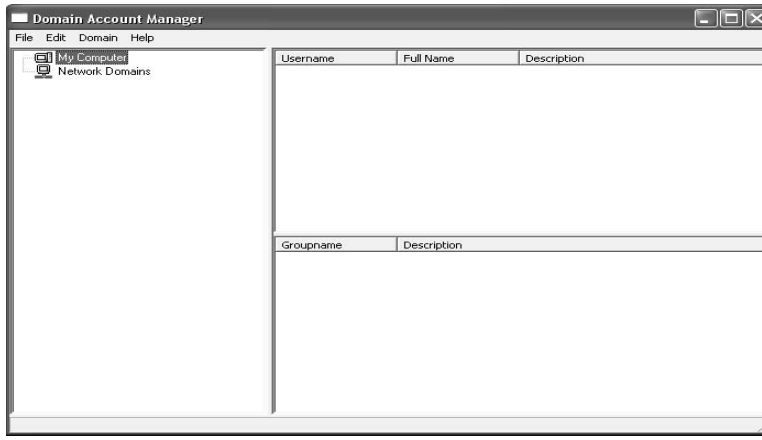


FIGURE 7.7

attempting to access the Internet. This would be the shared variable engine attempting to make itself available. Do not be alarmed that your machine has acquired a virus of some sort.

7.5 SHARED VARIABLE DOMAINS

The shared variable service uses the concept of domains to control access to shared variable services. The NI Security Service coordinates user logins and access to NI Services. It is a strong recommendation that the NI Security Services be used when deploying shared variables across any network.

The NI security apparatus is built on the concept of users, groups, and domains. A domain is a set of users and account policies. It is possible for users to have different policies applied in different domains. In order to create a domain, from LabVIEW select Tools-> Security-> Domain Account Manager to bring up the Manager. Figure 7.7 shows a machine with a fresh installation of LabVIEW — no domains exist on the local machine. The domain manager displays domains specified on the local machine and any known domains on the network. In order to create a domain, right click on local computer and create a domain.

The first dialog box to appear when a new domain is created is the Administrator password. Administrator is a unique account name, and its usage should be kept to a minimum. A machine can have a single domain. Once the password is created, the Administrator account is established, and the domain name is entered; you have to log in to the domain to do anything else. Creation of the machine's domain will be done as a "Guest" user, and the Guest user does not have authority to do anything with the machine's new domain. Log in as Administrator and add accounts for the user base. Once the domain is established, only accounts with administrator privileges will be permitted to modify anything in the Domain Account Manager. Figure 7.8 shows a local machine with a new LabVIEW Advanced Programming Techniques domain created. Only two predefined accounts exist for the domain, as mentioned,

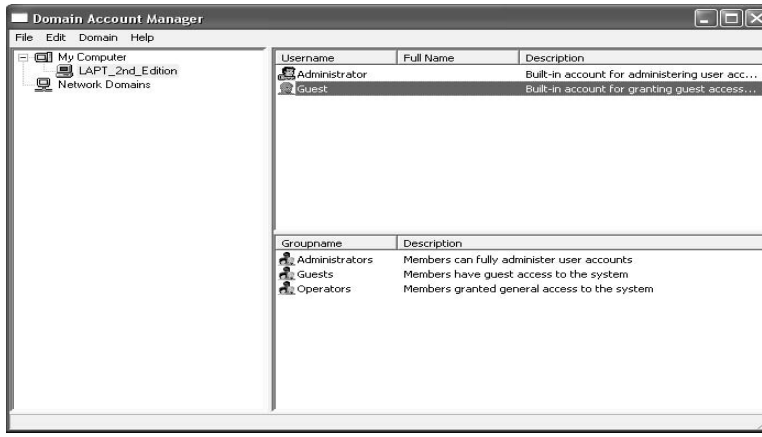


FIGURE 7.8

Administrator and Guest. In general, it is poor security practice to allow for only these two accounts. Every user should have a unique user account for the domain. In the case of the two predefined accounts, leaving everybody as a user allows everybody access to the entire NI security system which largely defeats the purpose of the system.

Once the domain is created, right click on it; selecting Properties shows the base configuration options for a domain. Figure 7.9 shows the general properties that can

FIGURE 7.9

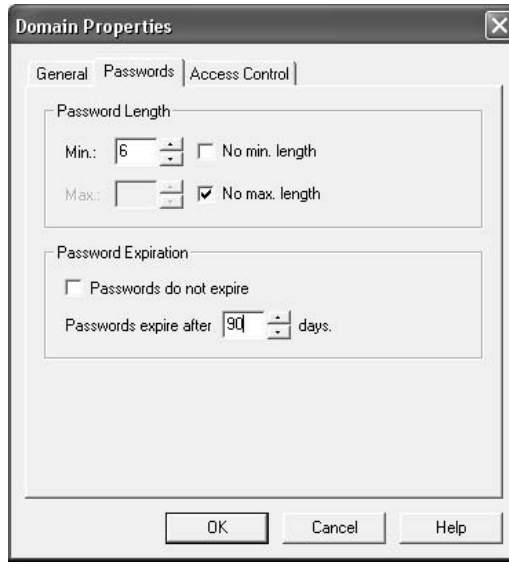


FIGURE 7.10

be configured consisting solely of the name of the domain. The hosting machine cannot be changed from this tab; to move the domain to a new machine, you will need to destroy it and recreate it on the target machine. The middle tab shows password policy configuration for the entire domain. Figure 7.10 shows the policy options available. It is recommended that password expiration policies be set for all domains. Depending on your employer or customer, guidelines on password expiration may already be defined. NI security settings should be configured to match any local IT requirements.

The last tab on the domain properties dialog is access control and is shown in Figure 7.11. This tab provides two lists, a list of machine-granted access and a list of machines denied access. Only one access list can be active at a time, so either this configuration is granting access to specific machines or denying access to specific machines. In general, we do not use this tab. If there is a security concern, it should be resolved with the machine's firewall, which will deny access to everything on the system instead of just NI-specific items.

In order to add a user to the domain, right click on the domain and select New User. Figure 7.12 shows the dialog box that is displayed. Once the user is given a name and description, a new password must be created before the user can be assigned membership to any groups. A base install of LabVIEW has a minimal set of security settings. The DSC Module permits for further security policy applications. Users are fairly self-explanatory. Any individual entity interfacing to the system is a user. Two built-in users are defined: the Administrator and Guest. Typically, each person using the system should be given a unique user account to the system.

Groups have three defined categories: Administrators, Operators, and Guests. It is possible to define additional groups. Additional groups with a basic LabVIEW

FIGURE 7.11

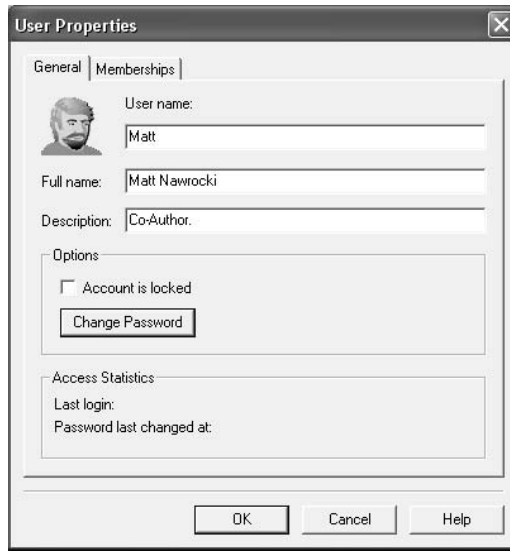


FIGURE 7.12

install do not add much value other than to provide logical groupings of users. The DSC Module adds some features that allow for additional security policies to be configured.

The security differences between the groups are largely minor. Administrator group members have access to security elements of the system. Operators can do everything with the exception of security, which includes starting and stopping variable libraries and adding variables to existing libraries. Guests surprisingly have a fair amount of permissions in the system by default, which include adding variables to the shared variable engine. There is no security configuration available in a base installation of LabVIEW. Tightening up the system would require either Lookout or the DSC Module.

Okay, so we have gone out and created a domain, added users, and assigned group membership to the users. The tie-in to the shared variable engine is now regulated by user permissions. Opening up the variable manager now requires a login and the security policies of the user are applied to limit what the user can do. Publishing variable libraries through the LabVIEW project will also use the security polices defined. Note that guest accounts have permission to publish variables to the engine by default.

7.6 PITFALLS OF DISTRIBUTED APPLICATIONS

This section will provide some information on issues that can be encountered when developing a distributed application; it is not entirely specific to the shared variable itself, but as shared variables can be used across machines on different networks, you may encounter some of the same issues.

This section is not intended to scare off development of a distributed application, but it is intended to raise a level of awareness as to what can be expected in the general realm of distributed applications. The shared variable abstracts a great deal of complexity around the network, but it cannot entirely absolve the developer of everything.

First and foremost, do not expect communications between computers to be 100% reliable every time, all the time. Links between machines do drop, packets get lost, multiple copies of the same packet may arrive, and packets may not arrive in the order they were sent. In general, a small network with a few switches will not see many instances of message duplication or out of order arrival; larger networks distributed across different ISPs, however, can. Applications built for wide area networks (WANs) need to consider link reliability. The shared variable engine is going to provide "best effort" in delivering updates on its distributed data, but the network is not always going to cooperate.

In the WAN scenario, the application must consider how sensitive the data is to time. If the application is not very time sensitive, then remote applications can work on stale data, and there is little to be concerned about. When time sensitivity is important, and data is not arriving on time, then errors need to be logged, alarms need to be sent out, and exception handling routines need to be engaged. One minor detail to consider on reporting errors: If the remote application is unable to communicate with the shared variable engine, then it probably can't send a network

message to that server with an error message. A machine that is collecting error information should not be the one running the shared variable engine.

The shared variable engine on a remote machine cannot notify a remote application that it cannot communicate with. If the remote application must have current data, timestamps need to be evaluated. Real time environments that are not validating correct information in the required time frames should be logging errors.

The shared variable engine uses Netbios, and Netbios does not always work well when machines are not regulated by a domain controller. In the event applications are operating in an environment without a domain controller, register the shared variable engine by IP address or name.

Distributed networks that are split across an internet service provider (ISP) may run into particular problems. Many ISPs use Dynamic Host Configuration Protocol (DHCP) to assign IP addresses to nodes. A DHCP system assigns addresses for set periods of time — called an address license. Once this license time expires, the node needs to renew its address license, and it frequently will be assigned a new address. In such cases, one can use domain name services such as DynDNS to keep a constant name for a shared variable server. The firewall or router interfacing to the ISP needs to be configured to use a domain name service. Once the domain names are established, it will be possible to register the variable engine by domain name on the remote computer.

7.7 SHARED VARIABLES AND NETWORK SECURITY

Everybody needs a hobby, and some people choose these better than others. For reasons beyond the scope of this book, there is a large community of people with apparently nothing better to do than break into computer networks. Shared variables bring a lot of the power of networking to LabVIEW applications, yet at the same time, they also bring in some of the vulnerabilities. A full discussion of network security is beyond the scope of this chapter, but this section will outline some of the concerns a developer needs to be aware of when deploying a networked application.

Network security is essentially a “negative deliverable.” Users will figure out when network security has failed, but proving that network security is working can be much harder. The simple fact that everything works may mean network security is preventing access, or it could mean that nobody has attempted to break into the network. The first assumption a developer needs to make is that attempts to violate the network will be made, and machines need to be configured to accept this reality.

First steps are fairly obvious and have nothing to do with LabVIEW. Every machine in the network should have antivirus applications with current virus definitions. Software firewalls should be operating on each machine, and a hardware firewall should restrict access at each location your network connects to the Internet. For example, a distributed network application gathering data at several manufacturing facilities may be exchanging shared variables over the Internet. Each facility needs to be firewall-protected at the Internet access point. In general, we never recommend using shared variables across the general public Internet. If your employer or customer has an IT department, they should be consulted on a distributed application. The IT organization will probably want to know the nature of the data

being exchanged and the volume of data that will be exchanged, so they can verify that the bandwidth available in the network is sufficient for your application and other applications running in the system.

Next on the list is the use of wireless Ethernet. Almost nothing gives external access to a system better than an improperly configured wireless Ethernet router. The two main configuration items for security on wireless routers is encryption and MAC layer filtering. Both should be used.

Shared variables do not use encryption; any data sent over the Internet will be sent "in the clear," meaning any third party that can see the message exchange can see the data. If remote facilities need to be interconnected, they should be connected through a virtual private network (VPN). VPNs use encryption for data transfer between two remote networks and makes the two remote networks appear to be one private network. VPN selection, configuration, and operation is a bit large in scope to present in this book. The IT department of your employer or customer should have already installed a VPN; as mentioned above, it would be to your benefit to make sure they are prepared to support any bandwidth you require.

In some cases, the customer may not be network savvy. It is your responsibility as the developer to clearly communicate security-related requirements. In the event customers lose data to hackers, they will be upset, and it is unlikely they will blame themselves when it happens! In the event you find yourself in this situation, a statement of work and breakdown of costs should include basic security items such as virus software and definition update subscriptions, firewalls, and, where required, VPN gateways.

As an example, consider a scenario where a distributed application is simply monitoring and controlling the coolers for a dozen flower shops. The average florist does not have many impacts on national security, and it would seem that security restrictions should be fairly minimal. Customer records and financial information are not handled by this system, which would seem to put this system a bit lower on the security priority. However, security on this system is not negligible. If somebody managed to hack into the system, they could potentially wreak havoc. Potential attack scenarios would be denial of service and playback attacks.

One type of simple attack is to flood the variable engine or the hosting machine with excess packets. If the attacker identified the port range of 6001 to 6010 correctly, he or she could hit the engine itself with a flood of packets. There is also a range of packets in the 5001 to 5005 range used that can be identified with a packet sniffer. If a system startup is monitored, the port 2343 would also be noticed. When the variable engine receives a malformed message, it will silently discard it but still has to process the message to determine it needs to be discarded. This would still amount to a denial of service. Flooding ports 2343, 5001 to 5005, and 6001 to 6010 would tie down NI's security service which could prevent machines from registering to the domain.

As PSP is not encrypted, somebody with unauthorized network access could start off by monitoring data exchanges. Packets could be resent to the server in order to try to define their purpose and monitor the responses. In the event the server engine is not set to any level of security, the hacker could try writing data to the system! Results of this could range from late night pages claiming the coolers are

running at 150° as an alarm condition to changing the set point of the cooler. The first scenario may sound somewhat amusing; the second scenario could result in the loss of inventory.

Before rendering a decision that the shared variable is too risky to deploy in a system, some common sense needs to be applied. It remains to be seen if the PSP protocol is hackable at this point. The PSP system has a lot more security built into it than many systems users would deploy. For example, if the example floral cooler control system were being built without the shared variable engine and instead was using custom written UDP packet exchanges, would the application validate the IP address where messages are coming in from? If not, then it would arguably be far less secure than a properly configured shared variable engine. The advice on this subject is pretty straightforward: Do not set a system up so that you will wind up finding out the expensive way. The shared variable engine greatly simplifies the act of distributing data, but security and access control need to be properly set up.

In order to minimize the potential of attacks, the following steps can be taken: First, designing distributed applications with central control may be undesirable. Remote clients should be designed such that they operate without the need of a central machine providing instructions. The cooler controls in this scenario should run the coolers regardless of the state of the network.

Firewall configurations could be set up to discard traffic from any machine not on a given list. The stock firewall in Windows XP Service Pack 2 allows for custom port permissions to be set which includes the ability to set a "scope." The scope can be set to allow only specific machines to pass traffic through the firewall on given ports. This is desirable because it eliminates any need to test the security of the NI system and reduces the processing requirements on the server. If the firewall rejects a packet at the entrance of the IP stack, the processing time required to route the doomed packet through the IP stack and into the NI security system will be eliminated.

Lastly, NI domains should be used. The standard LabVIEW installation does not provide much flexibility in the shared variable engine and domain security, but it can require somebody to log into the domain in order to interact with the engine.

If there are concerns that a system is under some type of attack, the tools mentioned to monitor the health of the engine can be used. If there are a dozen remote systems updating shared variables every minute, then the UDP traffic load is going to be low. The performance monitor can give an accurate count of how many UDP packets are arriving at any of the machines. If the counter values are through the roof, further investigation may be warranted.

Ethernet sniffers can be used to capture live data exchanges over a network for diagnosis. Ethereal is one off-the-shelf open source tool that does packet captures. This tool can be useful for monitoring traffic seen at an Ethernet interface. One issue to be aware of with Ethernet sniffers is they can generate very large trace files as networks tend to pass a lot of traffic that most users are not aware of. Systems with many traffic exchanges will generate tremendous amounts of traffic on their own; parsing an Ethernet trace file can become very time consuming.

In the event it becomes necessary to capture and analyze traffic, first start by identifying which IP addresses are the shared variable engine servers and which the clients are running on. Filters on the trace can be applied to minimize traffic

displayed. For example, if the server were also running a Web server, filtering out HTTP traffic could eliminate a large volume of packets. Netbios exchanges will be common between the clients and servers; there is no need for concern when you see these. A lot of short messages will be exchanged between clients and servers, and most of it will be in nonprintable characters. Things that should be looked for are IP addresses not on the known list attempting to access NI related ports (2343, 5001 to 5005, and 6001 to 6010).

7.7.1 LabVIEW Specific Security Issues

LabVIEW provides a domain system for regulating users and access to network resources. LabVIEW domains should not be confused with Windows Domains; they are not the same. LabVIEW domains define three groups of users, Administrators, Operators, and Guests. Each group definition grants a fairly broad level of access to the shared variable engine. The security settings on the variable manager should be among the first things to be modified after installation. Figure 7.13 shows the security settings panel. This panel is accessed from the Edit menu on the Variable Manager. The only settings that can be configured on a basic LabVIEW installation are the ability to modify security settings and to manage variables. Our recommendation is to add each group to the list and lock down engine access. Administrator group members should be the only ones able to modify the security settings and manage the variables. Guests should be set to deny access to both security and variable management. The operators group may need some flexibility. In a fully deployed system, they do not need to manage variables, whereas if a system is in development, this group may need the ability to add or remove variables and libraries from the system.

With a configuration set to deny both security and variable management, guests can still add items to the watch list, start the shared variable engine, and can acknowledge alarms. Guests cannot change security settings, add or remove vari-

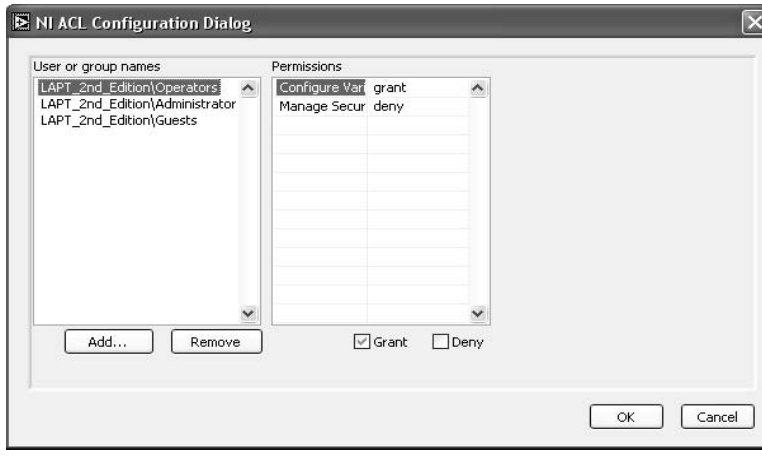


FIGURE 7.13

ables, start or stop processes, or add/remove libraries from the engine. The ability to start the shared variable engine is not much of a security risk; guests cannot stop the engine which would be a denial of service to other applications.

From a security perspective, most users need guest access once a system is deployed. Even better from a security perspective would be to use the DSC Module to tighten down permissions on the guest account. Individuals with administrator access should be kept to a minimum with a clear reason as to why they need the ability to configure the security settings of the system. Developers with a role that requires adding items to the system can be put into the Operator category, but if there is no particular need to start and stop processes or modify the libraries, they should be located into the guest group. It is possible to change a user from one group to another as required, and this is generally an advisable path to take. When a deployed system is running, there needs to be only a few administrators, and many guests. In the event an operator password is compromised, libraries can be removed from the shared variable engine; this would be a denial of service attack on the system.

BIBLIOGRAPHY

http://zone.ni.com/devzone/conceptd.nsf/webmain/
5B4C3CC1B2AD10BA862570F2007569EF#4.

8 .NET, ActiveX, and COM

As many readers have known and suspected, there is much more to .NET and ActiveX than a LabVIEW interface. .NET and ActiveX are relatively new technologies that wrap around several other technologies, namely OLE and COM. This chapter gives a general outline of .NET, ActiveX, COM, and OLE technologies and how they may be applied by the LabVIEW programmer.

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

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

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

Microsoft.NET is a general-purpose development platform that is used to connect devices, systems, and information through Web services. .NET relies on communications between different platforms across the Internet using standard protocols. .NET provides for security, exception handling, threading, I/O, deployment and object lifetime management.

Following the overview of these technologies there will be a section of examples. These examples will show how to use .NET, ActiveX, and VI server. The examples will help illustrate the concepts described below.

8.1 INTRODUCTION TO OLE, COM, AND ACTIVEX

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

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

The release of Windows 2000 included the release of COM+, which included the functionality of COM components and the Microsoft Transaction Server (MTS). COM+ now automatically handles resource pooling, disconnected applications, event publication and transactions.

Support for ActiveX and COM development exists in the Microsoft Foundation Classes (MFC) and the ActiveX Template Library (ATL). As ActiveX users, we do not need to be concerned with implementation issues; all we care about is that properties and methods can be invoked regardless of what the developers need to do to make it work. If you choose to develop your own ActiveX controls, you will need to do some research on which development tools and libraries to use, such as Visual Basic or C++.

8.1.1 Definition of Related Terms

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

8.1.1.1 Properties and Methods

All ActiveX controls have some combination of Properties and Methods that are available for other applications to control. Properties are variables that are exposed to outside applications. Properties can be configured to be readable only. Writable ActiveX control maintains a list of properties that may be accessed by external applications.

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

8.1.1.2 Interfaces

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

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

8.1.1.3 Clients and Servers

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

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

8.1.1.4 In-Process and Out-of-Process

COM objects may be represented in one of two manners: DLLs and executable applications. A DLL is linked into an application when it is started up. A COM object implemented as a DLL is referred to as an in-process COM object. When an ActiveX control is loaded into the same memory space as LabVIEW, it falls under the complete control of LabVIEW. When LabVIEW exits, the ActiveX control must also exit, and the memory space it resides in will be reclaimed by the system. Chapter 9 discusses protected memory.

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

In general, communication with in-process COM objects is faster than out-of-process COM objects. When data crosses a process boundary, the operating system must become involved. As processes are switched into and out of physical memory,

different threads of execution take over the CPU. This procedure generates additional overhead. This does not happen with in-process COM objects. The fact that out-of-process communication occurs should not dissuade a LabVIEW programmer from using COM servers. This is something that developers will need to be aware of when developing time-critical applications.

One advantage out-of-process servers have is stability. When an application is started, it is given its own address space and threads of execution. In the event that one process crashes, the other processes will continue to execute. These other processes may experience problems because a peer crashed, but good exception handling will keep other processes alive to accomplish their tasks. It is possible to crash peer processes, but defensive programming can always mitigate that risk. Out-of-process servers can also be run on different machines using DCOM, which allows for the process to continue executing even if one machine crashes. Good examples that can take advantage of distributed computing are monitoring and production software.

8.1.1.5 The Variant

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

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

8.2 COM

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

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

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

8.3 OLE

We will demonstrate in Section 8.4 that OLE interfaces are, in fact, ActiveX objects. The reverse is not always true; some ActiveX objects are not OLE interfaces. OLE is an interfacing technology used for automation, the control of one application by another. Many programmers subscribe to the myth that ActiveX has replaced OLE, but this is not true. OLE is a subset or specialization of ActiveX components. In general, ActiveX will be more widely used. LabVIEW applications that control applications like Microsoft Word and Excel are using OLE, not ActiveX.

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

8.4 ACTIVEX

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

8.4.1 Description of ActiveX

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

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

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

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

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

8.4.2 ActiveX Definitions

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

8.4.3 ActiveX Technologies

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

ActiveX documents (previously called OLE documents) are the part of ActiveX that is involved in linking, embedding, and editing objects. ActiveX documents deals with specific issues relating to "document-centric" functionality. One example is in-place editing. When a user wants to edit an Excel spreadsheet that is embedded in a Word document, the user doubleclicks on the spreadsheet. Unlike previous versions of OLE documents, the spreadsheet is not opened in Excel for editing. Instead, Word and Excel negotiate which items in the toolbar and menus are needed to perform

the editing. This function of ActiveX allows the user to edit the sheet in Word while still having all the necessary functionality. Another example of ActiveX documents is Web page viewing. When someone viewing a Web page opens a file, like a PDF file, the file can be displayed in the Web browser without having to save the file and open it separately in a PDF reader.

ActiveX controls (which replace OCX controls) are reusable objects that can be controlled by a variety of programming languages to provide added functionality, including support for events. ActiveX scripting is a means to drive an ActiveX control. This is mainly used in Web page development. An ActiveX control is lifeless without code to operate it. Because code cannot be embedded in the Web page, another method of control is necessary. That method is scripting languages. There are two common scripting languages supported in Web pages that are ActiveX compliant. The first is JScript (a type of JavaScript) and Microsoft Visual Basic Scripting Edition (VBScript).

8.4.3.1 ActiveX Terminology

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

A property is the definition of a specific object's parameters or attributes. For instance, in our baseball example, the player would have a uniform property. The user would be able to define whether the player was wearing the home or road uniform.

An event is a function call from an object that something has occurred. To continue the baseball analogy, an event could be compared to the scoreboard. When a pitch is made, the result of the pitch is recorded on the scoreboard. The scoreboard will show ball or strike, depending on the event that occurred. Events, as well as methods and properties, occur through automation mechanisms. Events are covered in more detail in the following section.

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

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

When an object is embedded in a container, the object's data is stored in the same file as the container. If an Excel worksheet is embedded in a Word document, that data is not available to any other application. When the worksheet is edited, the changes are saved in the data stream within the Word document. With both linking and embedding, a visual representation of the data in the container is stored in the container's file. This representation, called *presentation data*, is displayed when the object is not active. This is an important feature because it allows the file to be

viewed and printed without having to load the actual data from the file or data stream. A new image of the data is stored after the object has been edited.

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

8.4.4 Events

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

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

8.4.5 Containers

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

The container provides extender properties, methods, and events. They are written to be extensions of the control. The developer using the control should not be able to tell the difference between an extender property and the control's actual property.

There are a few suggested extender properties that all containers should implement. These controls are Name, Visible, Parent, Cancel, and Default. The Name property is the name the user assigns to the control. The Visible property indicates whether the control is visible. The Parent property indicates what the parent of the control is. The Cancel property indicates if the control is the cancel button for the container. The Default property indicates if the control is the default button for the container. There are a number of additional extender properties that are available to be used.

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

8.4.6 How ActiveX Controls Are Used

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

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

8.5 .NET

Microsoft .NET is a general-purpose development platform. .NET's main purpose is connecting devices, systems and information through Web Services. Web Services allow access to reusable components or applications by exchanging messages through standard Web protocols. The use of standard protocols like Extensible Markup Language (XML) and Simple Object Access Protocol (SOAP) enable computers from many different operating systems to work together. In addition to providing a means for cross-platform interoperability, .NET allows the user to run distributed applications across the Internet.

There are some additional benefits of .NET applications. The .NET Framework, which is one building block of the .NET platform, includes components for security,

exception handling, threading, I/O, deployment, and object lifetime management. The integration of theses services in the .NET Framework makes application development faster and easier.

8.5.1 Description of .NET

Microsoft .NET platform consists of numerous components and subcomponents. There are many books dedicated to individual pieces of the .NET platform, so we will just touch on some of the basics. In order to utilize .NET in your LabVIEW applications, very little knowledge of the nuts and bolts of .NET is necessary.

The .NET platform has six main components. The six components are the operating system, .NET Enterprise Servers, the .NET Framework, .NET Building Block Services, and Visual Studio .NET. The operating system is at the lowest level of .NET. The operating system includes desktops, servers, and devices. The inclusion of devices reflects the continuing trend to distribute applications and functionality to personal devices such as cell phones and PDAs. At the center of the .NET platform is the .NET Framework. The .NET framework consists of base classes to support Web Services and Forms. .NET provides a set of framework classes that every language uses. This will allow for easier development, deployment, and reliability. The .NET framework is built on the Common Language Runtime (CLR). The CLR is described further in the next section. At the highest level is Visual Studio .NET (VS.NET). VS.NET supports Visual Basic, Visual C++, Visual C#, and Visual J# for application development.

You may be asking yourself if .NET is supposed to be easier to develop and distribute, and has additional security and networking functionality, why would anyone still use COM? The question of what .NET means to COM's future is a controversial question. Well, first, let me assure you that COM is not going to become unsupported tomorrow. Microsoft has stated that the next Windows operating system will still support COM. There has been significant development of COM objects and applications. That is not going to be discarded. That being said, Microsoft does recommend that new application development be done using .NET. In order to be able to support existing ActiveX controls as well as to make the transition to .NET easier, you are able to call ActiveX objects in .NET.

8.5.2 Common Language Runtime

The Common Language Runtime (CLR) is the foundation for the .NET Framework. The CLR is a language-neutral development and execution environment that uses common services to perform application execution. The CLR activates objects, performs security checks on the objects, performs the necessary memory allocations, executes them, and then deallocates the memory used. The objects the CLR operates on are called Portable Executable (PE) files. The PE files are either EXE or DLL files that consist of metadata and code. Metadata is information about data that can include type definitions, version information, and external assembly references. The CLR also supports a standard exception-handling mechanism that works for all languages.

8.5.3 Intermediate Language

Intermediate Language (IL) is also known as Microsoft Intermediate Language (MSIL) or Common Intermediate Language (CIL). All .NET source code is compiled to IL during development. The IL is converted to machine code when the application is installed or at run-time by the Just-In-Time (JIT) compiler by the CLR. The IL supports all object-oriented features including data abstraction, inheritance, and polymorphism. The IL also supports exceptions, events, properties, fields and enumeration.

8.5.4 Web Protocols

There are several Web protocols utilized in .NET including XML, SOAP, HTML, and TCP/IP. XML provides a format for describing structured data. This is an improvement over Hypertext Markup Language (HTML) in that HTML is a fixed format that is predefined. XML lets you design your own markup languages for limitless different types of documents.

Simple Object Access Protocol (SOAP) is a simple, XML-based protocol that is used to exchange structured data on the Web. SOAP is the main standard for passing messages between Web services and the Web service consumers. SOAP defines the format of the XML request and responses used for the messaging. Through SOAP, objects can talk to each other across the Internet, even through firewalls. This helps eliminate an issue developers have had using distributed COM (DCOM). Because the IP address information was included in the messaging in DCOM, communications could not go through firewalls without configuration.

8.5.5 Assembly

The assembly was introduced in .NET. The assembly is similar to a compiled COM module. The assembly will be a DLL or EXE file. The assembly contains the IL code as well as a manifest that stores information on assembly name, version, location, security, files that are in the assembly, dependent assemblies, resources, exported data types and permission requests. A single module assembly has everything within one DLL or EXE. You can also use the assembly linker to create a multimodule assembly. The multimodule assembly can consist of many module and resource files.

An assembly can be private or public. The private assembly is a static assembly that is used by a specific application. A private assembly must exist in the same directory as the executable. A public or shared assembly can be used by any application. The shared assembly must have a unique shared name. All assemblies must be built with a public/private key pair. The public/private key pair will make the assembly unique. This can then be used by the CLR to ensure the correct assembly is used.

8.5.6 Global Assembly Cache

In the COM world, the shared library (DLL) used by the application is registered in the Windows Registry. When another application is installed on the computer, there is a risk of overwriting the DLL that is being used with another version that

does not support all the features that are needed. This is a common issue for COM programmers. .NET eliminates this issue by ensuring that the executable will use the DLL that it was created with. In .NET the DLL must be registered in the Global Assembly Cache (GAC). The GAC is a listing of public assemblies on your system. This is similar to the registry for COM objects. In the GAC the DLL must have unique hash value, public key, locale, and version. Once your DLL is registered in the GAC, its name is no longer an issue. You can have two different versions of the same DLL installed and running on the same system without causing problems. This is possible because the executable is bound to the specific version of the DLL. A .NET assembly stores all references and dependencies in a manifest. You can install, uninstall, and list assemblies in the GAC using the assembly registration utility (gacutil.exe).

8.6 LABVIEW AND ACTIVEX

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

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

8.6.1 The LabVIEW ActiveX Container

The ability to act as an ActiveX control container was first added to LabVIEW 5. By adhering to specifications for interacting with ActiveX controls, LabVIEW has become a client for these controls. It allows LabVIEW to utilize the functionality that the controls have to offer. To the user, an ActiveX control appears as any other control on the user interface. The operation of the control is seamless and unnoticeable to the end user. When used in conjunction with the available LabVIEW controls and indicators on the front panel, it becomes an integrated part of the application.

8.6.1.1 Embedding Objects

LabVIEW allows ActiveX controls and documents to be embedded on the front panel of a VI by using the container. Once the control or document has been placed

FIGURE 8.1

inside the container, it is ready for use and can be activated in place. Additionally, the objects' properties, methods, and events are made available to the programmer for use in the application.

A container can be placed on the front panel by accessing the ActiveX subpalette from the Controls palette. Then, to drop a control or document inside the container, pop up on the container and select Insert ActiveX Object from the menu. Figure 8.1 displays the dialog box that appears with the options that are available to the programmer once this is selected. Three options are presented in the drop-down box: Create Control, Create Document, and Create Object from File. The first option is used to drop a control, such as a checkbox or slide control, into the container. The other two are used to drop ActiveX documents into the container.

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

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

In both cases, the embedded object behaves as a compound document. Right-clicking on the object and selecting Edit Object launches the associated application.

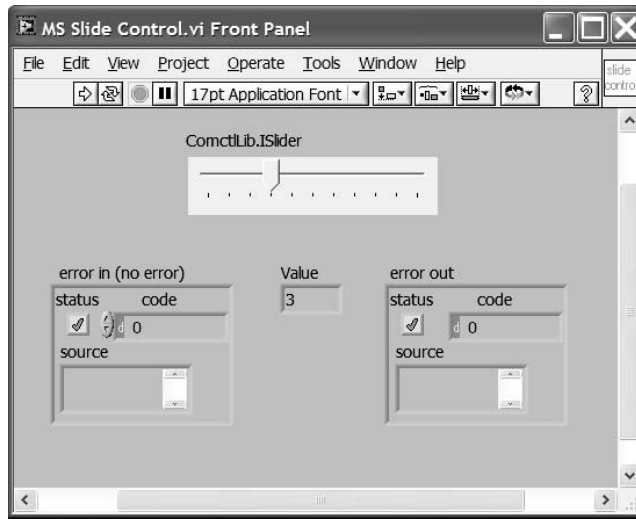


FIGURE 8.2

The user can then edit the document using the application. The automation interfaces that are supported by the application can be utilized from the code diagram after the object is placed inside the container. Once again, a refnum representing the container will appear on the code diagram.

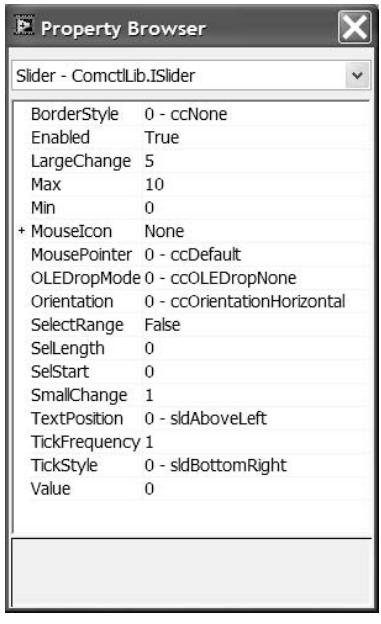


FIGURE 8.3

8.6.1.2 Inserting ActiveX Controls and Documents

This section will demonstrate how to insert and use ActiveX documents and controls within LabVIEW's container. Figure 8.2 shows the front panel of a VI with the Microsoft Slider Control, Version 6.0 (SP4), inserted into the container following the procedure described in the previous section. This control is similar to LabVIEW's built-in Horizontal Slide Control.

The properties of the slide control can be modified by right-clicking on it and selecting Property Browser. Figure 8.3 illustrates the window that appears for setting the slide's properties. There are options for altering the orientation, tick style, tick frequency, and mouse style of the control. Essentially, every

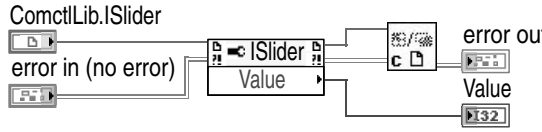


FIGURE 8.4

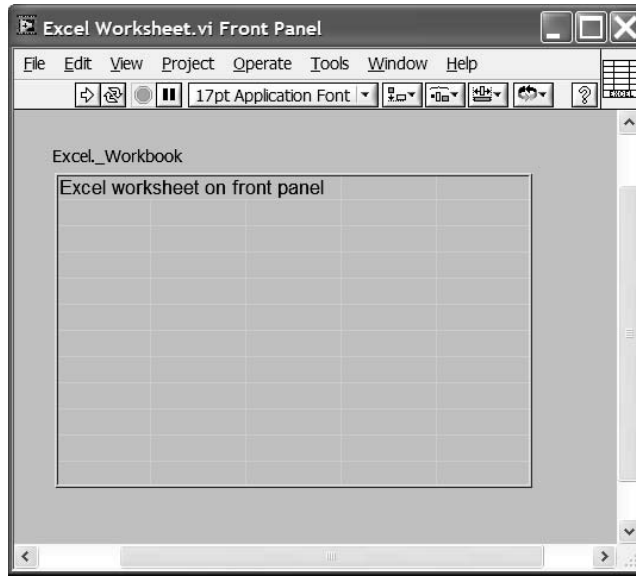


FIGURE 8.5

ActiveX control defines properties for which you can either set or retrieve the current value.

Figure 8.4 displays the code diagram of the same VI. Note that the ActiveX control appears as a refnum on the diagram. This is a simple example in which the Property node, detailed further in Section 8.6.2.2, is used to retrieve the value of the control set from the front panel. With LabVIEW's ActiveX container, numerous other controls can be integrated into an application. The programmer is no longer limited to the built-in controls.

ActiveX documents are embedded into LabVIEW's container by either inserting an existing file or creating a new document with an application that exposes its services. Figure 8.5 illustrates the front panel of a VI with a Microsoft Excel worksheet dropped into the ActiveX container. This was done by choosing Create Document from the drop-down menu, then selecting Microsoft Excel Worksheet from the list of available servers. The text shown was entered in the cell by right-clicking on the worksheet and selecting Edit Object to launch Microsoft Excel. Alternatively, you can pop up on the container worksheet and select Edit from the Document submenu. Figure 8.6 shows the same VI, but the Excel sheet is white with grid lines, which is the way you are probably used to looking at it. This was

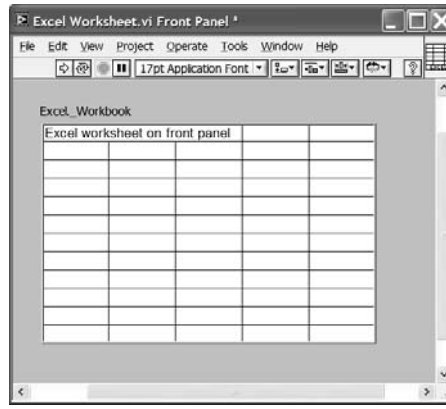


FIGURE 8.6

done by editing the object and filling the cells with white and selecting the gridlines. Once the Excel sheet is updated you can just close it, and the changes will be reflected on the VI front panel. It is not possible to just use the paintbrush to change the color as was done in earlier versions. The container refnum can now be utilized programmatically to interface with the worksheet's properties and methods. It is possible to use the properties to change the color programmatically as well.

8.6.2 The ActiveX Palette

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

8.6.2.1 Automation Open and Close

The function of Automation Open is to open a refnum to an ActiveX server. This refnum can then be passed to the other functions in the palette to perform specific actions programmatically. To create the Automation Refnum, first pop up on the Automation Open VI to choose the ActiveX class. Select Browse in the Select ActiveX Class submenu to see a list of the available controls, objects, and interfaces that LabVIEW has found on the system. Figure 8.7 shows the dialog box that appears with the drop-down menu for the type library, and the selection box for the object. LabVIEW gets information on the server's objects, methods, properties, and events through the type libraries. Additionally, a Browse button lets the programmer find

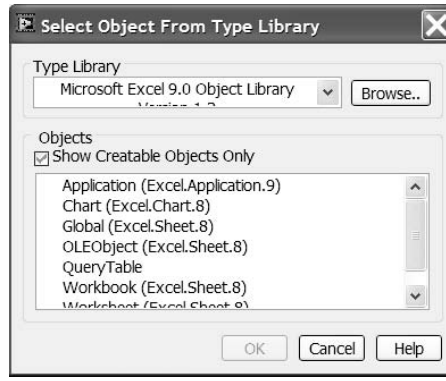


FIGURE 8.7

other type libraries, object libraries, and ActiveX controls that are not listed in the drop-down box.

Once you have found the object that you want to perform automation functions on, select OK on the dialog box, and the refnum will be created and wired to Automation Open. If you wire in a machine name, a reference to an object on a remote computer will be opened using DCOM. If it is left unwired, the reference will point to the object locally. DCOM objects can be instantiated only from the Automation Open VI.

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

8.6.2.2 The Property Node

The Property node is used to get or set properties of an ActiveX object. A refnum must be passed to the Property node to perform the Get or Set action on the object. The automation refnum, either from Automation Open or the refnum created from inserting a control into the front panel container, can be passed to the node. Note that this is the same Property node that is available in the Application Control subpalette to read or write properties for an application or VI. When performing application control, however, a reference must first be opened to an application or VI using the Open Application Reference function.

Once a refnum has been passed to the Property node, you can pop-up on the node and select a property that you wish to perform a read or write action on. The list of properties the object supports will be listed in the Properties submenu. If you wish to get or set more than one property, simply add elements from the pop-up menu, or drag the node to include as many as are needed. Then, select a property for each element. To change, or toggle, the action from read to write (or vice versa), select the associated menu item by popping up on the node. Some properties are read only, so you may not be able to set these properties. In this case, the selection in the pop-up menu will be disabled.

FIGURE 8.8

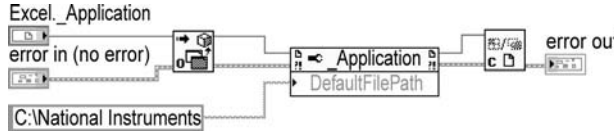


FIGURE 8.9

Figure 8.8 is an example in which the Property node is used to get the default file path that Microsoft Excel uses. Automation Open is used to open a reference to Microsoft Excel; Microsoft Excel 9.0 Object Library Version 1.3 was selected from the type library drop-down box and Application (Excel.Application.9) from the object list. DefaultFilePath is one of the items that is available for the application in the Properties submenu of the node. C:\Documents and Settings\Matt Nawrocki\My Documents was returned as the default file path that Excel uses when Save or Open is selected from the File menu. Alternatively, this property can easily be used to set the default file path that you want Excel to look in first when Open or Save is selected. First, you must pop-up on the Property node and select Change to Write from the menu. Then a file path can be wired into the node. This is depicted in Figure 8.9, where C:\National Instruments is set as the default path.

8.6.2.3 The Invoke Node

The Invoke node is the function used to execute methods that an ActiveX control makes available to a client. The number of methods that a particular control offers can vary depending on its complexity. Simple controls such as the Microsoft Slider Control, shown earlier in Figure 8.2, have only five methods. Complex objects like Microsoft Excel, may have upwards of several hundred methods available. Virtually all actions that can be performed using Microsoft Excel can also be performed using automation.

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

Figure 8.10 shows the code diagram of an example using the Invoke node to execute a method in Microsoft Excel. As before, Automation Open is used to open a reference to Microsoft Excel. Then, the refnum is passed to the Invoke node, and

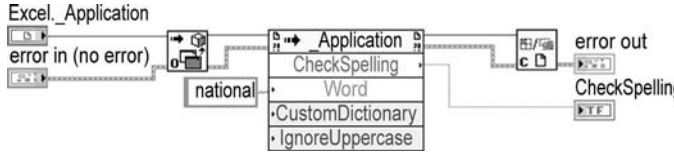


FIGURE 8.10

CheckSpelling is selected from the Methods submenu. When the method is selected, the input and output parameters appear below it. The CheckSpelling method checks the spelling of the word passed to it using Excel's dictionary. If the spelling is correct, a "true" value is returned. If the spelling is incorrect, a "false" value is returned. The only required input for this method is Word; the rest are optional. In a similar manner, you can utilize the services offered by any object that supports automation. This is a very effective way to realize code reuse in applications.

Complex ActiveX servers arrange their accessible properties and methods in a hierarchy. This requires the programmer to use properties and methods of a server to get to other available properties and methods. Applications such as Microsoft Excel and Word operate in a similar manner. After you have opened a reference to one of these applications, the Invoke node and Property node can be used to get to other properties and methods that are not directly available. The first reference opened with Automation Open is also known as a "creatable object." Note that when you are selecting the ActiveX class for Automation Open, the dialog box (shown in Figure 8.7) gives you the option to show only the createable objects in the selection list. When the option is enabled, LabVIEW lists those objects that are at the top of the hierarchy. What does this mean to a programmer who wants to use ActiveX and automation? For simple objects, a programmer is not required to know the details about the hierarchy to use them. They have relatively few properties and methods, and are simple to use programmatically. When complex objects or applications are being used, however, a programmer needs to know how the hierarchy of services is arranged to achieve the desired result in a program. This means that you have to refer to the documentation on the server, or help files, to use them effectively. The documentation will help guide you in utilizing ActiveX in your applications.

Figure 8.11 illustrates the hierarchy of Microsoft Word through an example. This VI opens a Microsoft Word document, named test.doc, and returns the number of total characters in the document. First, a reference to Microsoft Word Application is opened using Automation Open. Then, Documents is the first property selected with the Property node. Documents is a refnum with a set of properties and methods under its hierarchy. The Open method is then used to open the file by specifying its path. Next, Characters is a property whose Count property is executed to retrieve the number of characters contained in the document. Observe that all references are closed after they are no longer needed. The Microsoft Word hierarchy used in this example proceeds as shown in Figure 8.12. Microsoft Word application is at the top of the hierarchy with its properties and methods following in the order shown.

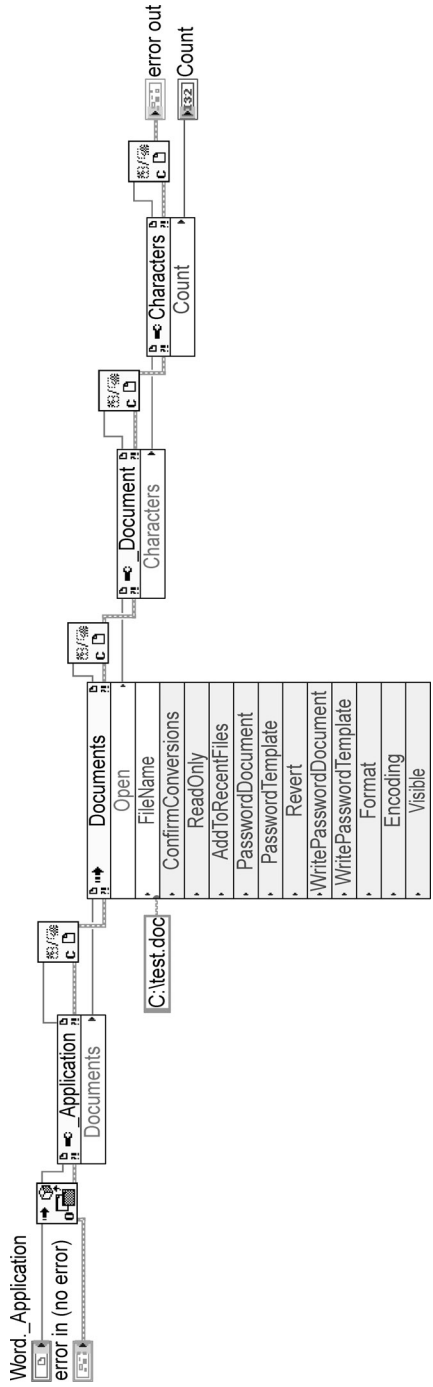

Word._Application
error in (no error)
_Application
Documents
Documents
Open
FileName
ConfirmConversions
ReadOnly
AddToRecentFiles
PasswordDocument
PasswordTemplate
Revert
WritePasswordDocument
WritePasswordTemplate
Format
Encoding
Visible
C:\test.doc
_Document
Characters
_Characters
Count
I32
Count
error out

FIGURE 8.11

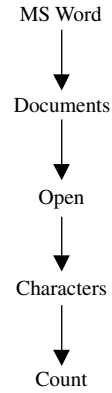


FIGURE 8.12

8.6.2.4 Variant to Data Function

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

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

Figure 8.13 shows an example using To G Data. The code diagram shown is a subVI in which the value of the active cell in a Microsoft Excel workbook is being read. Its caller, which opens a reference to Excel and the workbook, also passes in the Application refnum for use. As shown, the data type for the cell value is a variant. The actual data in the spreadsheet is a string, therefore, the variant passed from

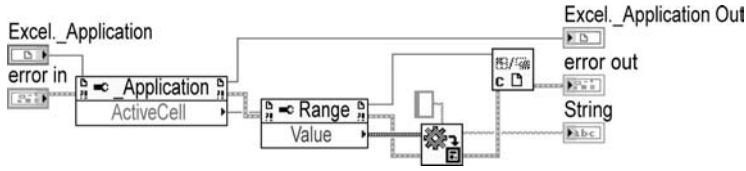


FIGURE 8.13

Excel must be converted to a valid LabVIEW data type using the Variant to Data function. An empty string constant is wired to the type input, and a string indicator is wired to its output. If the data in the workbook is a number, a numeric constant (integer or floating point) can be wired to the type input for conversion.

When an ActiveX object requires a variant data type as input, the programmer is usually not required to perform any conversion from the valid LabVIEW types. There is a function on the ActiveX subpalette that does convert to the Variant data type if needed. Look back at Figure 8.11, the example in which a Microsoft Word document was opened and the number of characters retrieved. The Open method required a file name of variant type. A string constant was wired to this input without any conversion. If you look closely, you will see a coercion dot, indicating that LabVIEW coerced the data to fit the type required.

8.6.3 Using the Container versus Automation

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

When you need to use ActiveX automation to work with applications like Microsoft Word or Excel, it should be done from the code diagram, without using the container. Use Automation Open to create a refnum by selecting the ActiveX Class to make use of the available services. Then, use Invoke node and Property node to perform needed actions, as shown in Figure 8.11. A front panel container was not used in that example. You may need to refer to documentation on the objects' hierarchy of services. On the other hand, if a specific document's contents need to be displayed and edited, then you can embed the document on the front panel and perform the actions needed.

8.6.4 Event Support in LabVIEW

Many ActiveX controls define a set of events, in addition to properties and methods that are associated with it. These events are then accessible to the client when they occur. In order to handle an event, the event must first be registered. The event can then be handled by a callback. This is significant because LabVIEW can interface to properties, methods, and events to fully utilize all of the services that ActiveX objects have to offer. This section will explain and demonstrate how to use the Event functions available in the ActiveX (or .NET) subpalette.

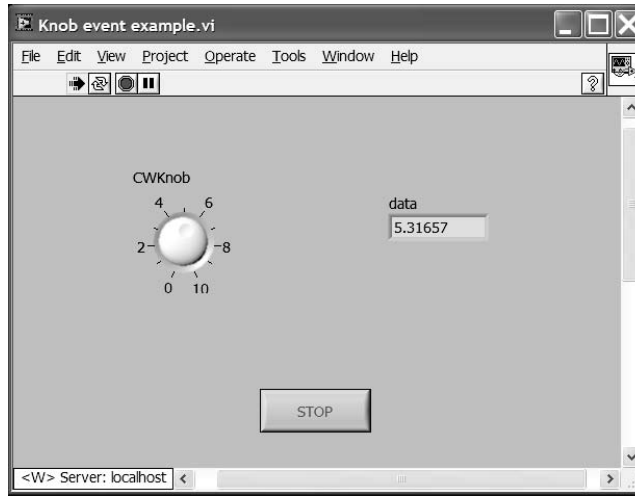


FIGURE 8.14

8.6.4.1 Register Event

For our example on using ActiveX events, we will put a knob control on the front panel of a VI and display the current value in a digital indicator. There will also be a stop button to stop execution. The front panel is shown in Figure 8.14. The first step in working with ActiveX events is to insert the control in a container or use the automation refnum to call an Active X object. For this example we will place a Component Works Knob in an ActiveX container on the front panel. This will create a reference on the code diagram. Now we need to insert the Register Event Callback Function on the code diagram. In order to be able to select an event to operate on you will need to wire the Knob control reference to the Event input on the Register Event Callback function. You might initially think that the reference should go to the Register Event Callback reference input, but the callback reference is a separate reference that does not need to be wired. Now that the reference is connected to the Event input, you will be able to click on the down arrow to select the desired event. For our example we want to display the value of the knob in a digital indicator as the knob is changed. To do this we need to select the PointerValueChanged event.

The Register Event Callback function should now have three inputs listed: the Event input, the VI reference, and an input for user parameters. The VI reference input, which is a link to the callback that will handle the event, will be discussed in the next section.

8.6.4.2 Event Callback

The Event Callback is a VI generated to perform any operations when the event occurs. For our example we want to display the digital value of the knob when it is changed. To do this we need to change the value of the data indicator in the main VI. Because this piece of data will need to be updated in the callback, the information

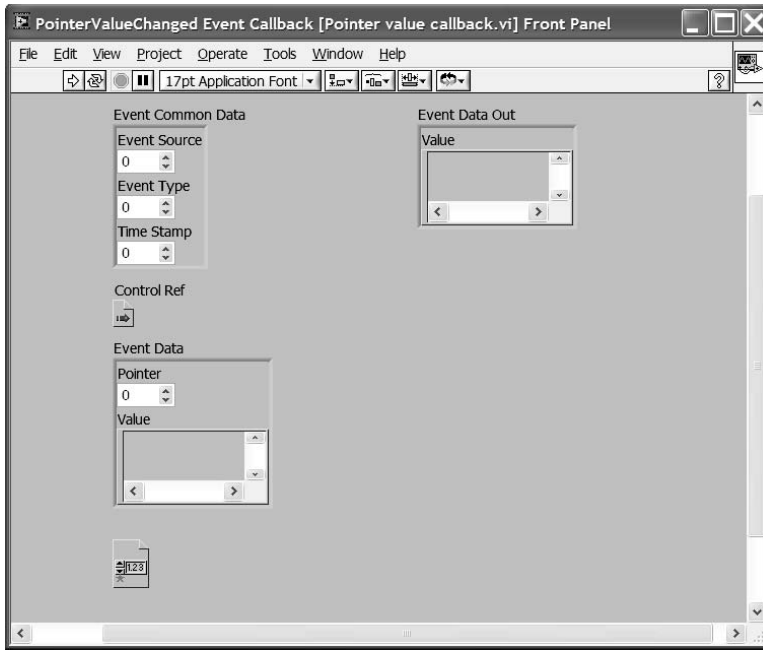


FIGURE 8.15

will need to be wired to the user parameters input on the Register Event Callback function. To do this you need to create a VI Server Reference for your indicator. This is done by right clicking on the indicator and selecting Reference from the Create option. This reference can now be used to access the properties and methods of the indicator.

Once you have wired the knob control reference and the user data input to the function, you can create the callback. To create the callback you right click on the Callback VI reference input and select Create Callback VI. A new VI will open with several controls and an indicator on it. What shows up on the callback will change based on the event selected and the information wired to the User Data input. The callback VI front panel is shown in Figure 8.15.

The Event Common Data control will provide a numeric value representing what the source of the event was (LabVIEW, ActiveX or .NET). The Event Type specifies what type of event occurred. For a user interface event the type is an enumerated type control. For all other event types a 32-bit unsigned integer is returned. The time stamp indicates when the event occurred in milliseconds. The Event Data control (and sometimes indicator) is a cluster of event-specific parameters that the callback VI handles.

To be able to update the value of the digital indicator in the main VI we need to first get the current value of the control. This is done by unbundling the value parameter from the Event Data control. What is contained in the value control is the current value stored as a variant. We must convert the variant to a number using the Variant to Data function. To be able to update the data indicator we will access

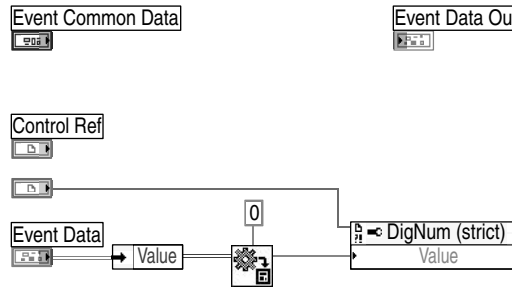


FIGURE 8.16

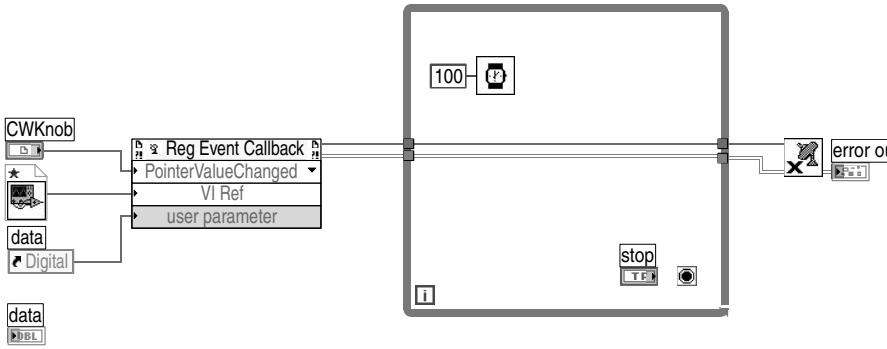


FIGURE 8.17

the value property from the data indicator reference and pass the current value. The final callback code diagram is shown in Figure 8.16.

Now that the callback has been created we can finish our main VI. In order to continually monitor the knob we insert a While loop (with a delay to free up system resources) on the code diagram. We also insert a stop button to exit the loop when we are done. Finally, we need to unregister our event. You notice that nowhere on the main VI do we do anything with the data indicator. This is all handled through the callback, the indicator will update whenever the event occurs. The code diagram for the main VI is shown in Figure 8.17.

8.6.5 LabVIEW as ActiveX Server

LabVIEW has both ActiveX client and server capabilities. This means that you can call and use external services inside LabVIEW's environment, as well as call and use LabVIEW services from other client environments such as Visual Basic. This functionality enhances the ability to reuse LabVIEW code by being able to call and run VIs from other programming languages. It also gives you flexibility to use the language that best suits your needs without having to worry about code compatibility.

When using LabVIEW as an ActiveX server, the Application and Virtual Instrument classes are available to the client. Consult the online reference for details on the methods and properties that these classes expose to clients. To use LabVIEW

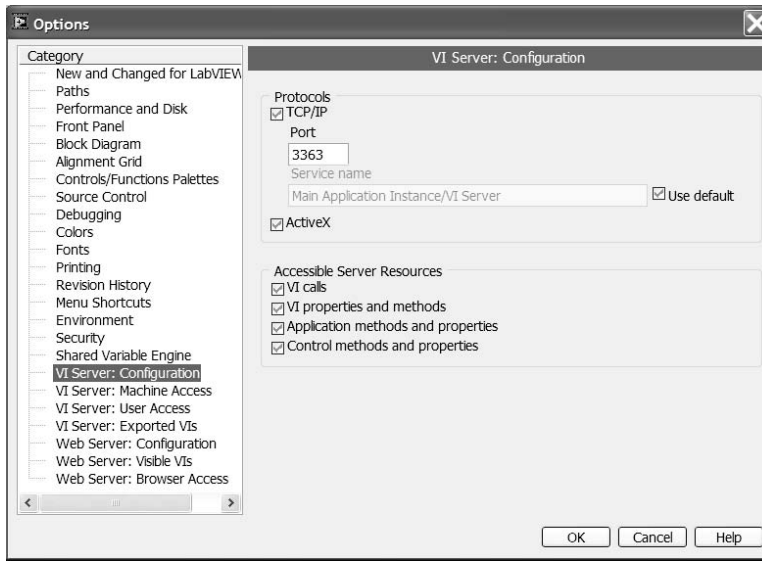


FIGURE 8.18

as an ActiveX server, you must first ensure that this functionality has been initiated. Select Options from the Tools menu, and VI Server: Configuration as shown in Figure 8.18. Check the ActiveX protocol selection and the appropriate Accessible Server Resources as needed. You can allow VI calls, VI properties and methods, Application methods and properties, and Control methods and properties. The online reference also has an ActiveX example to demonstrate how this feature can be used. It is an example of a Visual Basic macro in which a VI is loaded, run, and its results retrieved for further use.

8.7 LABVIEW AND .NET

As with using ActiveX in LabVIEW, you do not need to have in-depth knowledge of .NET to be able to utilize the controls and functionality that .NET provides. Now that you have some exposure to properties and methods, you will find that implementation of .NET is fairly easy. LabVIEW can be used as a .NET client to access the properties, methods and events of a .NET server, but LabVIEW is not a .NET server. Other applications cannot directly connect to LabVIEW using .NET.

8.7.1 .NET Containers

One part of .NET that can be used through LabVIEW is the .NET control. The .NET control is similar to an ActiveX control. LabVIEW allows you to embed the control on the front panel. The .NET control will appear like a standard LabVIEW control and will operate seamlessly when its attributes are controlled in your application. In order to use a .NET control on the front panel of your application you will need

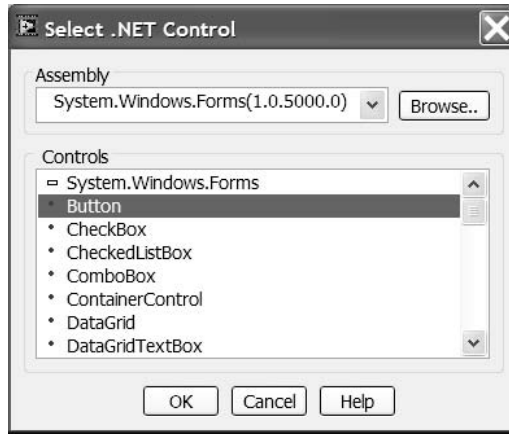


FIGURE 8.19

to use a container. The .NET container is available to place on the Front Panel from the .NET/ActiveX palette.

Once you have placed a container on the front panel you can right click on the container and choose Insert .NET control. This will cause a dialog box to come up that will allow you to search for the control you want to use in your application. The dialog box is shown in Figure 8.19. For this example we will select the radio button control. This control is contained in the System.Windows.Forms assembly.

Now that your control is embedded in your front panel you can start to program the application to utilize the control. One way to start is to use the class browser. Using the class browser you can view the properties and methods for an assortment of items including VI Server, ActiveX and .NET objects. To start you will need to launch the class browser from the tools menu. Once the browser launches you can select the object library. From here you select what type of object you are looking for and then either select an object that is already loaded or to browse for your object. Once the object is chosen, in this case the System.Windows.Forms, you can select the Class. For our example we choose the RadioButton class. Now you can view all the available properties and methods. The class browser is shown in Figure 8.20. Notice in the class browser that there are several methods with the same names. The Scale method and the SetBounds method are listed twice. In the case of the first Scale method the inputs are Single dx and Single dy. The second Scale method has a single input named Single ratio. In .NET you can have a single property or method name with different inputs. If you select a property or method you can place the item on the code diagram by selecting create or create write at the bottom of the dialog and then placing the node on the diagram.

For our example we are just going to have a loop that will monitor the radio button status. When the radio button is clicked the loop will stop. The value of the radio button will be displayed in an indicator. As we already have the radio button on the front panel, we will use the class browser to select the checked property. This property will output a Boolean value. Once the property button is placed on the code

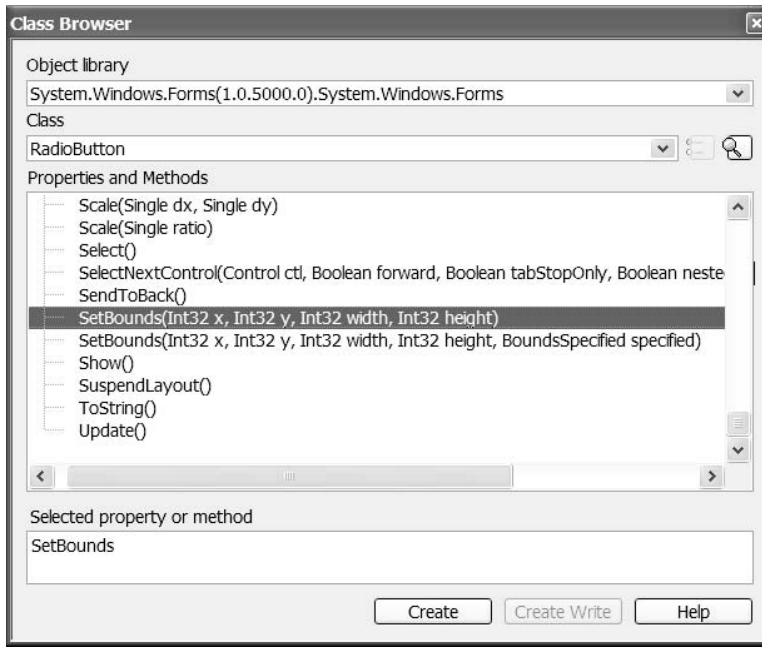


FIGURE 8.20

diagram you can wire the output to the conditional terminal of the While loop. The code diagram is shown in Figure 8.21. The front panel of the running VI is shown in Figure 8.22.

Obviously this is a simple example, but the same method would be used for larger applications.

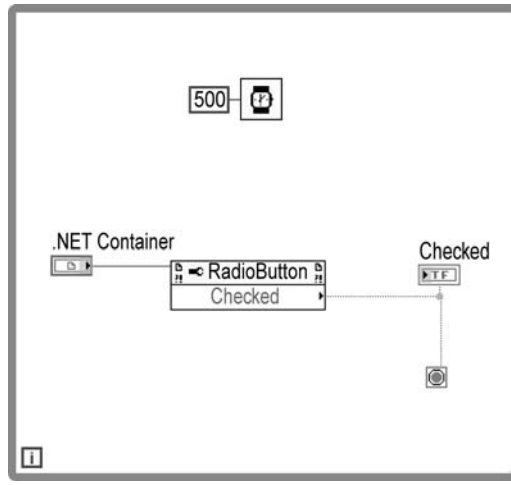


FIGURE 8.21

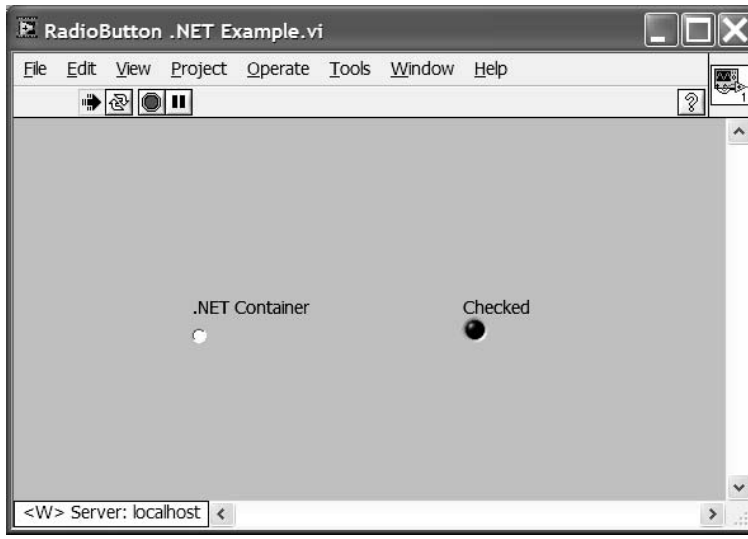


FIGURE 8.22

You must have the .NET framework installed. As discussed earlier, the .NET framework is the core of .NET, and is required to be able to be able to create and use .NET objects. The framework can be downloaded from the Microsoft Website.

8.7.2 .NET Palette

The .NET palette contains most of the same functions as the ActiveX palette. There is a Property node, Invoke node, register and unregister events, close reference, and static VI reference. There are a couple of new functions to the .NET palette. The first is the constructor node. This is similar to the automation open in ActiveX programming. The constructor is used to create an instance of a .NET object. There is also a pair of functions for converting a reference to a more specific or general class. The common properties have been discussed in depth in the ActiveX section; here, we will cover only the constructor node and the class conversion functions.

As mentioned, the constructor node is used to create an instance of a .NET object. The first step is to place the constructor node on the code diagram. Once the node is placed on the diagram, a dialog automatically opens. If you close the dialog box, or if you decide to select a different constructor, you can right click on the constructor node and choose Select Constructor. The top entry of the dialog is the assembly. For this example we will select the System assembly. Now you can go through the second window, which shows the creatable objects. We have scanned through the list to get to the System.Timers object. There are several subobjects under the main object. For this example the Timer object is selected. In the bottom window there are two available constructors. They are both timer constructors, but they have different inputs. Here, the Timer() constructor will be selected. The dialog is shown in Figure 8.23. Once the constructor is loaded on the diagram you can access the properties and methods.

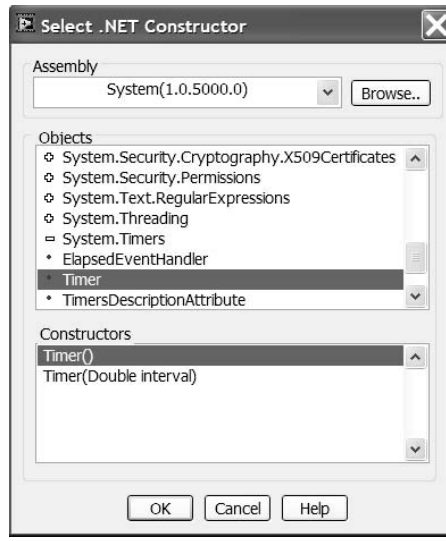


FIGURE 8.23

The class conversion functions are similar to the typecast function. They are used to convert one type of reference to another. In some cases, a reference generated through a property or invoke node does not yield the properties or methods needed. The reference might be a general reference that needs to be more specific. I know this sounds confusing, but bear with me. Let us say that you go to a full service gas station to get gas for your car. You pull up to the pump, tell the attendant you want a fill-up. At this point you have opened a reference for the generic gas class. The problem is you don't want just any kind of gas; you want to get unleaded gas. You need to convert your existing reference to a more specific reference, the unleaded gas reference. Now you can access the properties of the unleaded gas reference such as grade. This is what you are doing with the To More Specific Class. You are getting access to the properties or methods of a more specific class than the reference you currently have. There is an example of using this function to control an instrument through .NET in the examples section.

To use the To More Specific Class function you need to connect the current reference. There is also an input for the error cluster. Finally, there is an input for the Target Class. You can use a constructor node to generate a reference for this input. The output is the type of reference you need for the selected object.

8.8 THE VI SERVER

VI Server functionality allows programmers to control LabVIEW applications and VIs remotely. The functions available for use with the VI Server are on the Application Control subpalette. You must first ensure that LabVIEW is configured correctly to be able to use this capability. Figure 8.18 displays the VI Server configuration dialog box. Activate the TCP/IP protocol, then enter the port and enable the

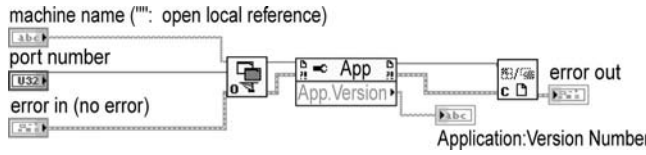


FIGURE 8.24

appropriate server resources you wish to use. In the VI Server: Machine Access menu item, you can enter a list of computers and IP addresses that you wish to allow or deny access to the VI Server. In the VI Server: User Access menu item, you can enter a list users and groups that you wish to allow or deny access to the VI Server. Finally, the VI Server: Exported VIs item allows you to create a list of VIs that you want to allow or deny access to remotely. The wildcard character (*) allows or restricts access to all VIs on the machine.

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

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

The code diagram in Figure 8.24 demonstrates how to make use of the VI Server. This example simply retrieves the version number of the LabVIEW application on a remote computer. First, a reference to the application is opened and the machine name and port number are wired in. Then the Property node is used to get the version number, and the reference is subsequently closed. The online help describes all of the properties and methods that can be utilized programmatically.

The Open VI Reference function is used to open a reference to a specific VI on either a local or remote machine. Simply wire in the file path of the VI to which you wish to open the reference. If the VI is on a remote computer, use Open Application Reference first, and wire the refnum to the Open VI Reference input. Figure 8.25 shows an example first introduced in the Exception Handling chapter. It opens a reference to External Handler.vi and sends it the error cluster information

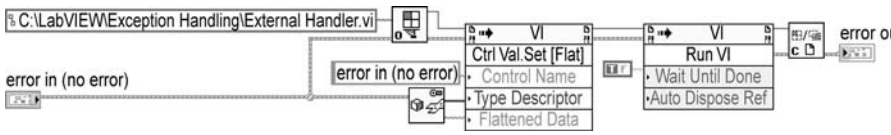


FIGURE 8.25

using the Set Control Value method. Then the External Handler is executed using the Run VI method. Finally, the reference to the VI is closed with Close Application or VI Reference.

8.9 ACTIVEX AND .NET EXAMPLES

In this section we will provide several examples that utilize ActiveX and .NET controls and automation. The material here serves two purposes. The first is to give you enough exposure to these technologies so that you will be comfortable using them in your own applications effectively. By employing the services offered by 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 that you may not have considered before.

8.9.1 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 which are using this control. It was desirable for Microsoft to have standard mechanisms for users to perform common tasks through user interface elements. 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 8.26 displays the code diagram of Common Dialog.vi. The Common Dialog control was placed in the front panel ActiveX container, and is represented

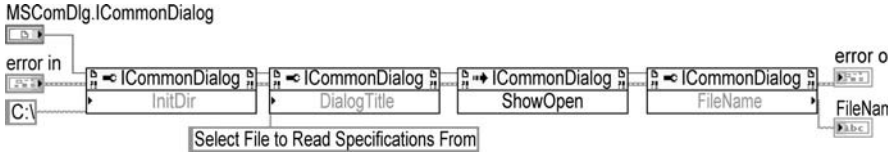


FIGURE 8.26

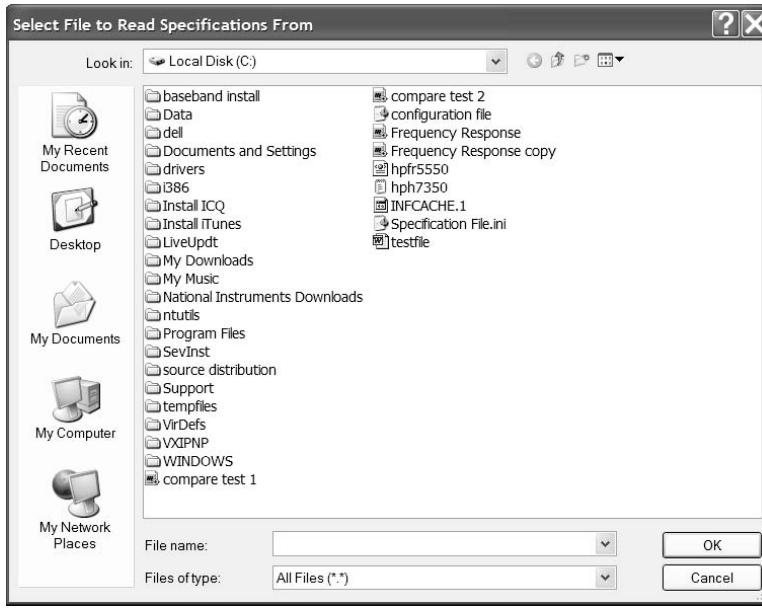


FIGURE 8.27

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 8.27 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. This being the first example presented, two separate property nodes have been used to simplify the block diagram.

8.9.2 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 and

.NET, 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.

As this is a user interface element, there are two different methods that we can use to create it: we can insert it into a container, or we can use the ActiveX Open. User interface elements are typically created with a 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 either an ActiveX or .NET container on the front panel. Once the container is placed on the front panel, right-clicking on the control allows you to insert an 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 numerous properties and methods. 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.

Minimum, Maximum and Value properties are used to drive the control itself. Minimum and Maximum 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

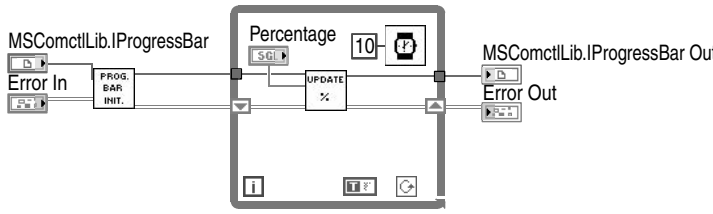


FIGURE 8.28

inputs do not need to be applied every time we configure a control. The code diagram for this VI appears in Figure 8.28.

8.9.3 Microsoft Calendar Control

This example uses the Microsoft Calendar Control, Version 9.0. 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 9.0. 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 properties. (Note that there is a .NET calendar control on the .NET palette that has many of the same properties and methods.)

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 background color of the control. 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 8.29. 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.

8.9.4 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. It allows you to navigate the Web programmatically through the COM interface and see the pages displayed on the front panel window. This control can be dropped into LabVIEW's front panel container. In LabVIEW 8, an icon for an ActiveX Web browser container is in the .NET & ActiveX palette. The resulting container on the front panel is equivalent to inserting the ActiveX object into an empty container.

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 online documentation or technical support for an operator. A simplified user interface is displayed in Figure 8.30 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 online support that is desired. When the 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 8.31 shows the code diagram of this user interface. After setting the Caption property of the CommandButton, an event queue is created for Click. The

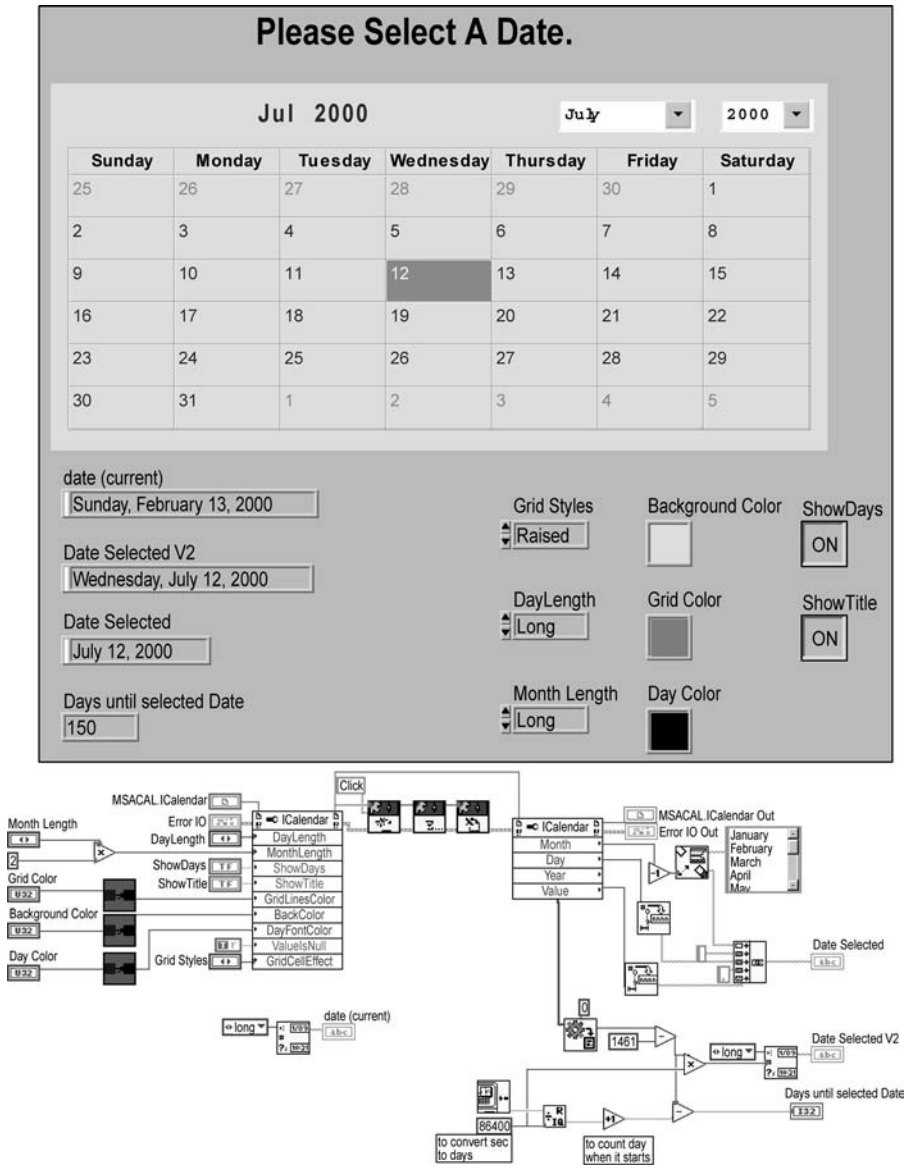

Please Select A Date.
Jul 2000
July
2000
Sunday Monday Tuesday Wednesday Thursday Friday Saturday
25 26 27 28 29 30 1
2 3 4 5 6 7 8
9 10 11 12 13 14 15
16 17 18 19 20 21 22
23 24 25 26 27 28 29
30 31 1 2 3 4 5
date (current)
Sunday, February 13, 2000
Date Selected V2
Wednesday, July 12, 2000
Date Selected
July 12, 2000
Days until selected Date
150
Grid Styles
Raised
DayLength
Long
Month Length
Long
Background Color
Grid Color
Day Color
ShowDays
ON
ShowTitle
ON
MSACAL ICalendar
Error IO
DayLength
ShowDays
ShowTitle
Grid Styles
ICalendar
DayLength
MonthLength
ShowDays
ShowTitle
GridLinesColor
BackColor
DayFontColor
ValueIsNull
GridCellEffect
Month Length
Grid Color
Background Color
Day Color
Click
ICalendar
Month
Day
Year
Value
MSACAL ICalendar Out
Error IO Out
January
February
March
April
May
Date Selected
long
date (current)
1461
long
Date Selected V2
Days until selected Date
86400
to convert sec to days
to count day when it starts

FIGURE 8.29

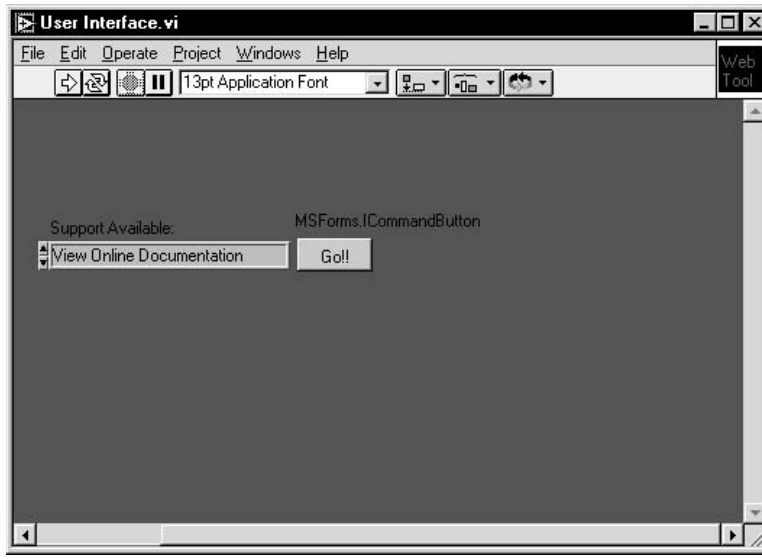


FIGURE 8.30

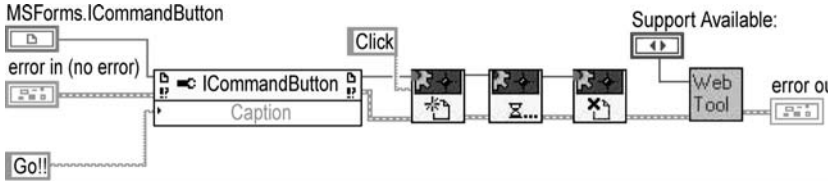


FIGURE 8.31

VI sleeps until 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 8.32 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 8.33. 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 online, the Done CommandButton is pressed to return to the main user interface.

Figure 8.34 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.

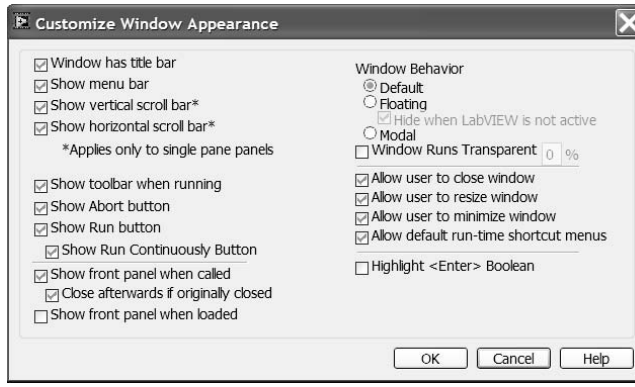


FIGURE 8.32

FIGURE 8.33

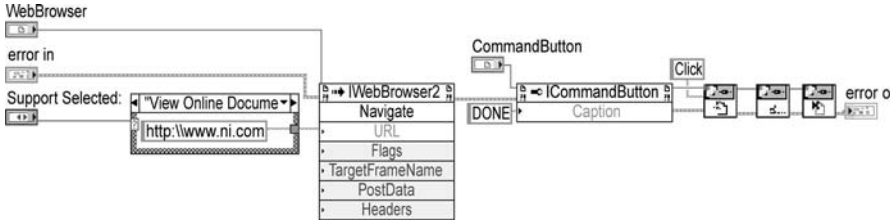


FIGURE 8.34

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.

8.9.5 Microsoft Scripting Control

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 may 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

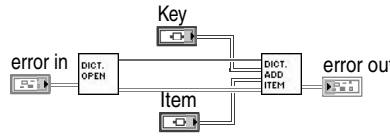


FIGURE 8.35

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 8.35.

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 it 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

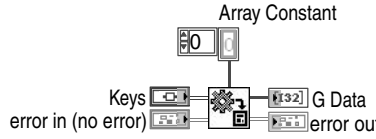


FIGURE 8.36

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 8.36. 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 autoindexing 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.

8.9.6 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 8.37 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

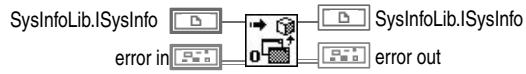


FIGURE 8.37

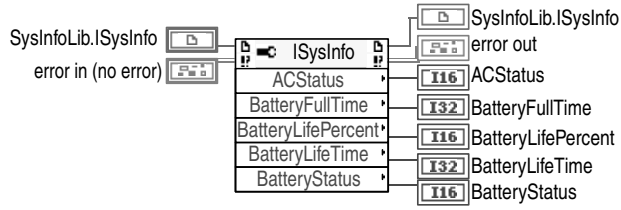


FIGURE 8.38

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 8.38 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 releases than most people realize. If you are doing hardcore 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.

Operating system properties available are the version, build number, and platform. “Platform” indicates which Win32 system is running on the machine. The

"Build Number" indicates which particular compile number of the operating system you are running. Actually, this property will not be particularly 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.

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 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.

8.9.7 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 8.39 shows the menu selection leading us to the status bar's properties. We can select

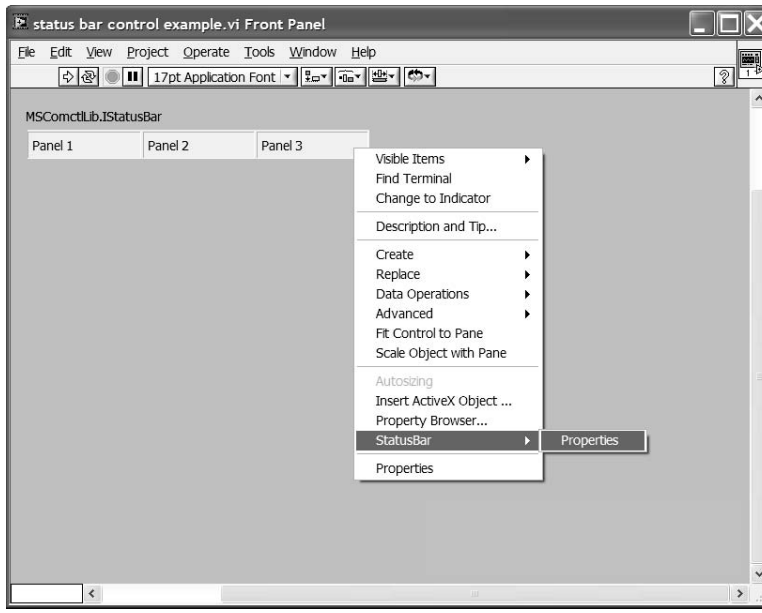


FIGURE 8.39

general properties to configure basic appearance information such as 2- or 3-D 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 8.40 shows the Tabs Configuration window for this control. Each panel is flexible in the sense that

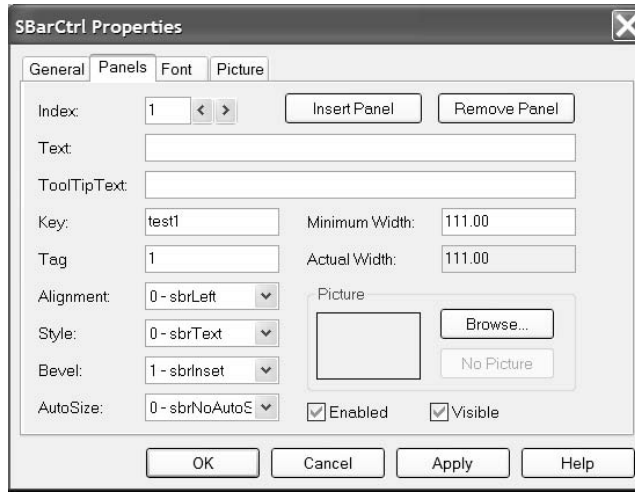


FIGURE 8.40

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 runtime.

In order to work with the control at runtime, 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, Item (Get), Item (Put Ref), 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 Remove method takes a variant argument similar to the Item method. This method removes the specified method from the status bar.

Item (Get) and Item(Put Ref) allow us to modify a single panel item. Both methods return a refnum to the specific panel. The Item (Get) 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(Put Ref) 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(Put Ref) method in our driver.

You may wonder why there are two different Item methods. The additional text in parentheses indicates that there are two different 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 text in parentheses makes the name different, so we are not violating the "one function, one signature" rule.

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,

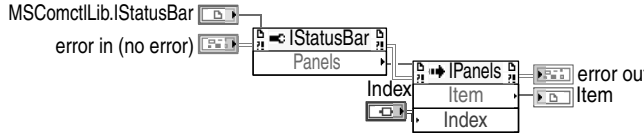


FIGURE 8.41

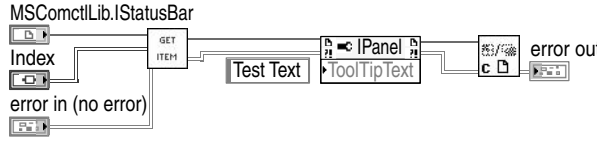


FIGURE 8.42

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 8.41.

The VI to set the Tool Tip text for an individual panel item is shown in Figure 8.42. 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 are available on the companion CD to this book.

8.9.8 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 function is now built into LabVIEW as the Tree Control; we will also discuss the ActiveX implementation for this control. 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 a 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 need to specify only 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 8.43.

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

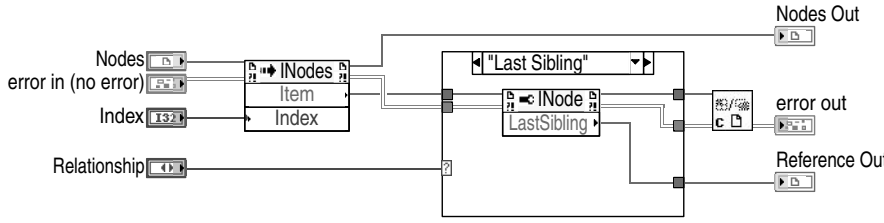


FIGURE 8.43

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 similarly 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.

8.9.9 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 Website.

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 follows the COM design methodology, and each of the branches in the hierarchy is 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.

8.9.9.1 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.

Programmers who are interested in using the Request object in their work will want to know that its properties include Status, Number, and Description. 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.

8.9.9.2 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 serves 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 Website, 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 can be useful for error-handling routines.

8.9.9.3 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 familiar 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.

8.9.9.4 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 going to need only 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; requiring programmers (including yourself) 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 need 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 need to instantiate the control only 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 Figure 8.44 and Figure 8.45. 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

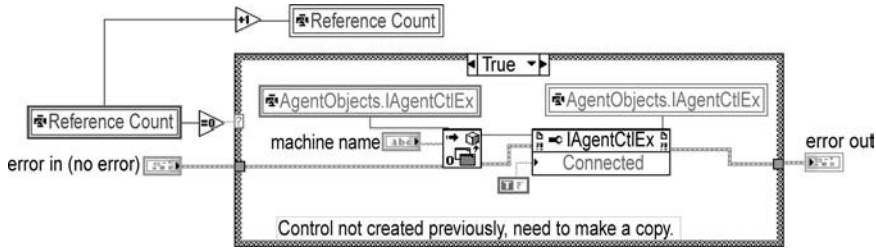


FIGURE 8.44

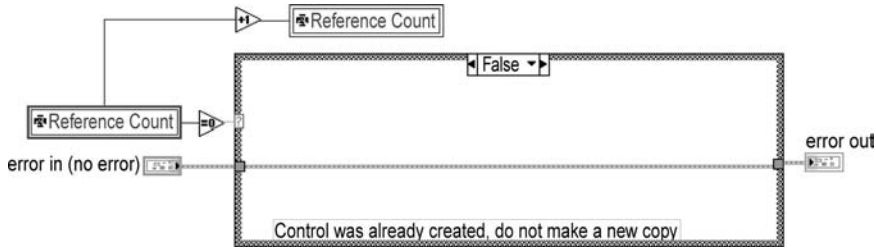


FIGURE 8.45

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. Visual Basic programmers who do not have an error cluster to work with need Request objects. This VI is shown in Figure 8.46.

It is important to note that we closed off only the Request object's refnum. We are not finished with either the control or the character refnums; in fact, we have

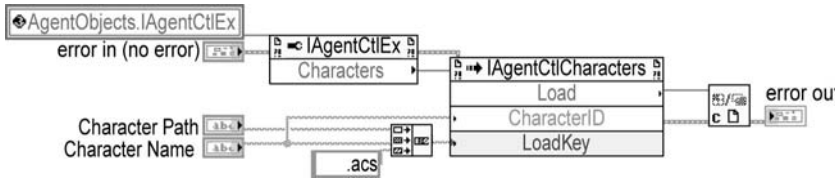


FIGURE 8.46

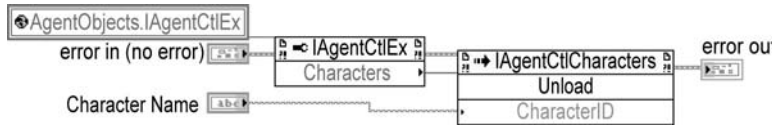


FIGURE 8.47

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 8.47. 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 needed only when passing instructions to specific characters. Programmers will need to pass only 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 8.48. 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 each 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 8.49.

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

FIGURE 8.48

FIGURE 8.49

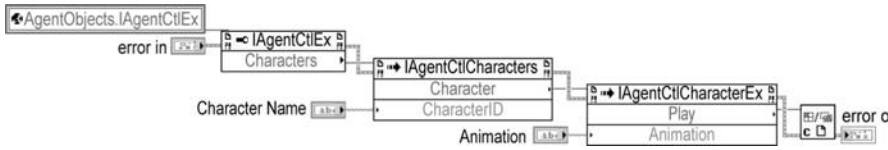


FIGURE 8.50

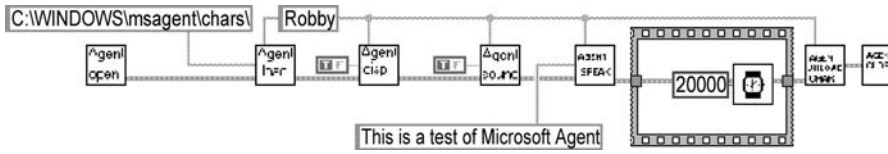


FIGURE 8.51

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 8.50. 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 8.51 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 midsentence!

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.

8.9.10 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 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_USERS, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE and HKEY_CURRENT_CONFIG. Application information is stored in the HKEY_CLASSES_ROOT hive, and generic user information is HKEY_USERS. 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 five 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 do 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.

8.9.11 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 2000 (Version 9.0) has two main creatable 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 whopping125 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.

After opening 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 8.52.

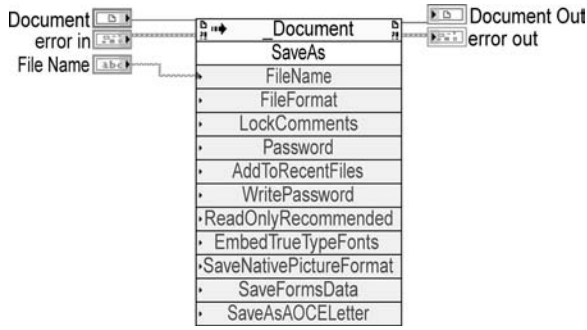


FIGURE 8.52

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 to do programmatically. 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. We will 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 need to use only 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 are included on the companion CD.

8.9.12 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 9.0 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 8.53 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

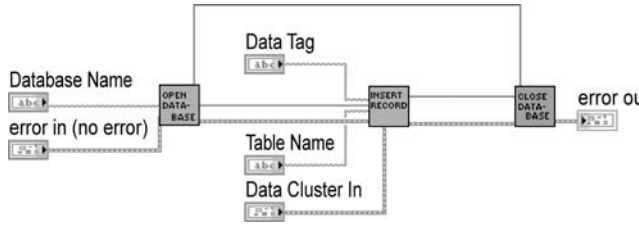


FIGURE 8.53

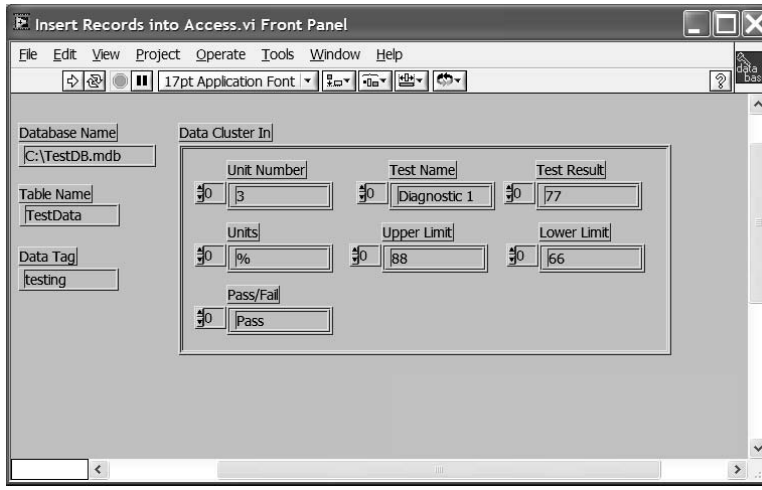


FIGURE 8.54

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 8.54. Each control in the cluster is an array, and hence 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 because the references to Access are opened and closed each time this VI is called. This VI can be used if you have an existing database, but you should be 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 8.55. Microsoft Access 9.0 Object Library Version 9.0 was selected from the Object Type Library list from the dialog window for selecting the ActiveX class. The Application object was selected from the list in the object library.

TABLE 8.1
Steps to Create a Database and Table in MS Access

Step 1. Launch Access application.
Step 2. Select New Database from the File pull-down menu.
Step 3. Select Blank Database, Name, and Save the Database using the file dialog box. TestDB.mdb is the name used in this example.
Step 4. Select the Tables tab, select New to create a table, and select Design View.
Step 5. Enter the field names as shown in Figure 8.56.
Step 6. Name and save the table. TestData is the name of the table used in this example. A Primary key is not needed.
Step 7. Close the database. LabVIEW will not be able to write to the database while it is open.

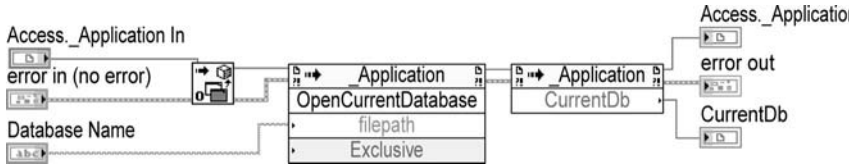


FIGURE 8.55

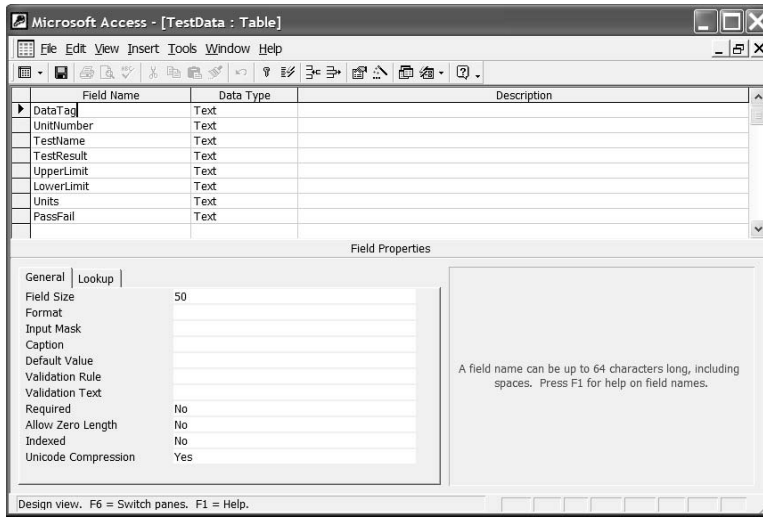


FIGURE 8.56

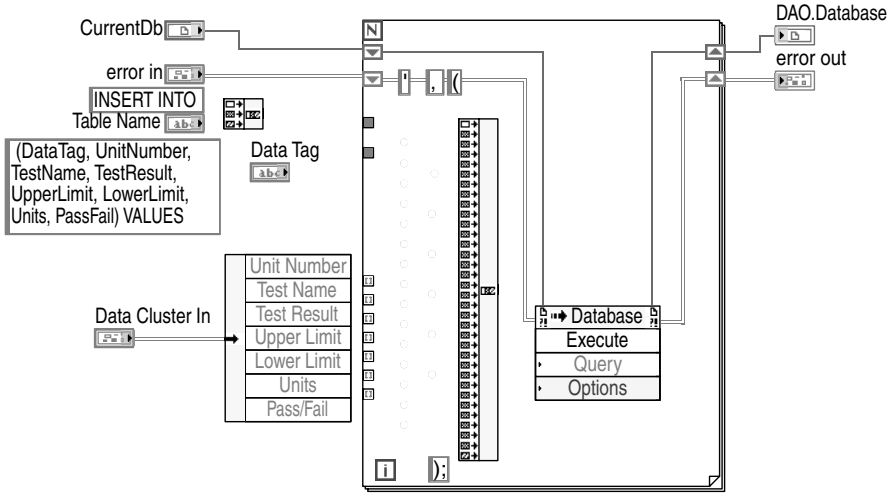


FIGURE 8.57

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 8.57. 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,

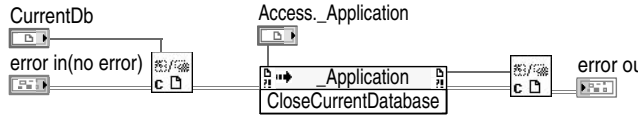


FIGURE 8.58

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 8.58. 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.

8.9.13 Instrument Control Using ActiveX

The examples to this point have shown how to insert objects in your code and to control other applications. We will now discuss a method for controlling a piece of test equipment using ActiveX. In reality, you are not really controlling the instrument directly; you are controlling an application that controls the instrument. For this example we will discuss the Agilent 89601A Vector Signal Analysis software.

The Agilent 89601A software is designed to demodulate and analyze signals. The inputs to the software can come from simulated hardware, the Advanced Design System (ADS), or from actual test equipment. The 89601A software can act as a front end for test equipment like the 89441 vector signal analyzer (VSA), the PSA series spectrum analyzers, the Infiniium oscilloscopes and 89600 VXI-based vector signal analysis hardware.

In this example the 89601A software is controlling an 89441 VSA. There is a lot involved in setting up this instrument for taking measurements on modulated signals. Here we are describing only one VI used to set the frequency parameters of the instrument. The first step is verifying there is no error input. An error input

will cause the code to exit without opening any references. In order to begin using the software you will need to open a reference to the 89601A. Once the reference to the application is open you will need to open the measurement property. Using the measurement property reference you can open a reference to the frequency property. Through this property all the necessary frequency properties such as center frequency, span, number of points and window are now available.

The center frequency and span properties require only numeric inputs. The number of points is also a numeric input, but for the ease of programming and configuration, the LabVIEW input for the number of points is an enumerated type control. The enumerated type contains the valid frequency point values: 51, 101, 201, 401, 801, 1601, and 3201. This enumerated type is converted to a string to pull out the selected value, and then is converted to a number for the property.

The window property requires an enumerated type input. The text of the enumerated type is not very user friendly. For this reason we created a separate enumerated type control for user input that will be converted to the appropriate 89601 enumerated type. The input options are Flat Top, Gaussian, Hanning, and Uniform. The index of the selected user enumerated type control is used as an input to an index array function. The other input for this function is an array constant containing the matching 89601 enumerated type values. The output is what is written to the Window property. Finally, the references for application are all closed before exiting. The code diagram of this VI is shown in Figure 8.59.

As you can see, there is no difference between automating functions with a Word document and automating the control of the 89441 VSA using the 89601A software. Obviously this was a simple example. To fully automate control of the instrument there are many settings such as the measurement type, sweep mode and averaging. All functions can be automated in the same way the frequency settings were written.

8.9.14 Instrument Control Using .NET

In the last example we discussed a method for controlling a piece of test equipment using ActiveX. For newer equipment and software applications, the interface will likely be .NET. One such example of an instrument that can be controlled through a .NET interface is the Agilent ESG series vector signal generator. As always, standard communications to the instrument are available through GPIB and LAN connections. In order to extend the capabilities of the ESG to be able to capture waveforms, playback waveforms and fade signals, the processing for the generator was moved to the PC. Through the use of a PCI card installed on the host computer and a cable connecting the PCI card to the generator, the ability to manipulate the waveform data is now possible on the PC using Agilent's Baseband Studio software.

For this example we will be focusing on using the signal generator to fade a radio frequency (RF) signal. In RF receiver design and in faded signal analysis it is important to be able to simulate real-world scenarios without costly field testing. The results will also be more repeatable. You may be asking yourself what a faded signal is. One example is when a signal from a cellular base station is sent to a receiver (cell phone); the signal can take many paths to get there. This occurs when

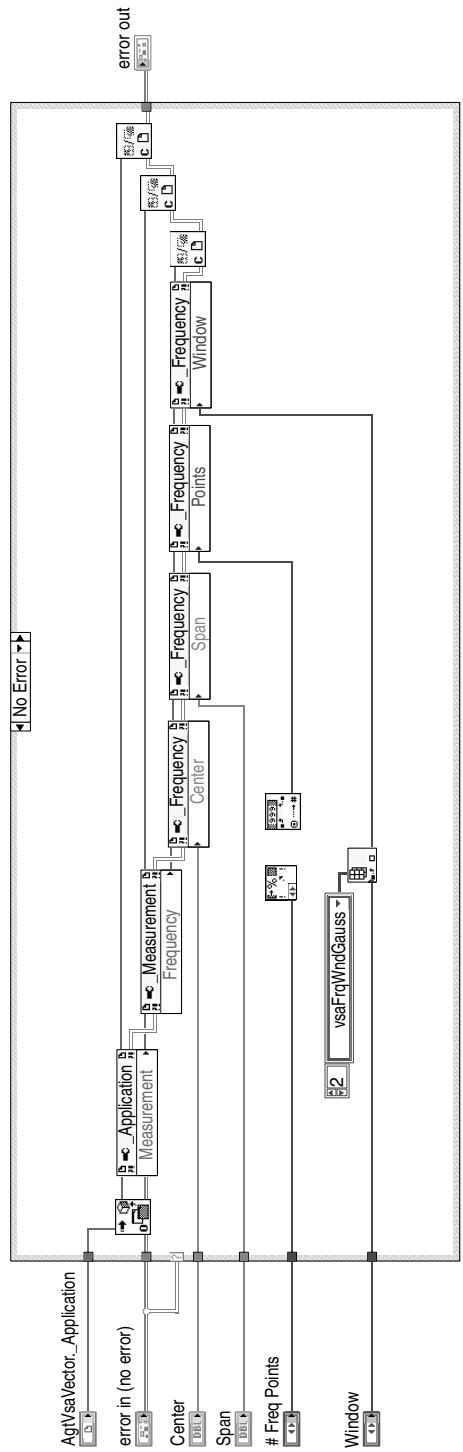

error out
No Error
_Application
Measurement
_Measurement
Frequency
_Frequency
Center
_Frequency
Span
_Frequency
Points
_Frequency
Window
vsaFrqWndGauss
AgtVsaVector._Application
error in (no error)
Center
Span
Freq Points
Window

FIGURE 8.59

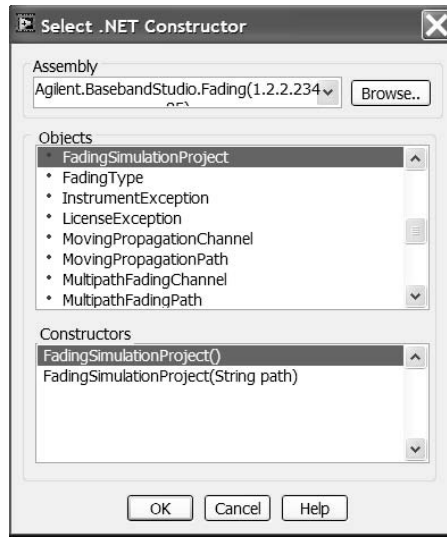


FIGURE 8.60

obstructions are encountered in the path between the tower and the receiver. The receiver will receive all these different paths causing the received signal amplitude and phase to fluctuate when the signals add constructively and destructively. There is also the matter of the receiver being in motion. These factors are all a part of fading. For this example, knowledge of fading is not needed, other than for the terminology used.

The first step in automating the faded signal generation is to create the Project. In .NET a constructor is used to open a reference to the application. A constructor node needs to be put on the code diagram. The constructor can be found in the .NET palette. When the constructor node is placed on the code diagram a dialog is opened for selecting the constructor. This dialog box is shown in Figure 8.60. Agilent.BasebandStudio.Fading is chosen from a list of registered .NET assemblies. The objects for this assembly are then populated below. The Fading Simulation Project is selected from the Objects section. When the object is selected, the available constructors show up in the bottom box in the dialog. Here we select the Fading Simulation Project.

Now that the constructor is linked we can select from a list of methods to call. We could open an existing project, save a project or apply the settings. To start we choose to open a new project. From here we are doing all of our operations on this new fading simulation project. The open project code is shown in Figure 8.61. An

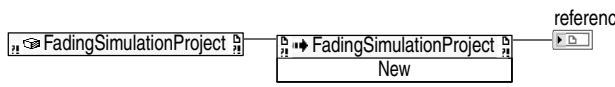


FIGURE 8.61

issue that was observed when using this fader application with LabVIEW involved the reference timing out. If there is no activity between the LabVIEW application and the fading software for an extended period of time, the reference would close. In order to get around this problem, code was inserted at the main level of the application to query the fader enable state every 5 minutes. This effectively kept the connection open until the application was closed out.

From this point on, the programming required is similar to manually setting up the profile in the application window. To start we need to add a fading channel and define the paths. For this we must first add the channel. The Add Channel method is derived by first creating a channel property from the Simulation property. To add the channel we first need to use a constructor node to create a Multipath Fading Channel reference. This reference is used as an input to the Add Channel method. It is also an input to the Fading Channel paths property node. For this example there are 12 possible paths for the channel. We will need to use another constructor node to create the path reference. The code required to creating a new fading channel with 12 paths is shown in Figure 8.62.

Now that the paths have been created, we will need to enter information into each path (if used). To do this we will first need to obtain a reference to a specific path. This is done by opening a Fading Simulation Channel reference and selecting (get_Item) the desired channel. The resulting reference allows us to create a path reference. With the path reference we can select (get_Item) the specific path we want a reference to. The resulting reference is a generic path reference. In order to be able to select the needed properties and methods, the reference needs to be typecast to a more specific class. There is a function to do this in the .NET palette. The function takes the generic reference as an input. The other input is the type you are converting to. This is generated using a constructor node. The multipath fading path constructor is chosen. This will creates the correct reference for modifying the needed path parameters. The code to obtain the specific path is shown in Figure 8.63.

Now that we have a reference we need to configure the settings of the faded signal. Settings for speed and loss are entered as needed to create the desired profile. The remaining setup will not be discussed here as the implementation is fairly straightforward. The parameters for the faded path can be set using properties and methods just like any other .NET or ActiveX application.

8.9.15 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 performed 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 earlier, OLE interfaces are a superset of ActiveX interfaces. Both technologies rely upon COM's specification.

We will be controlling LabVIEW through Microsoft Word. Visual Basic for Applications (VBA) is built into several Microsoft applications.

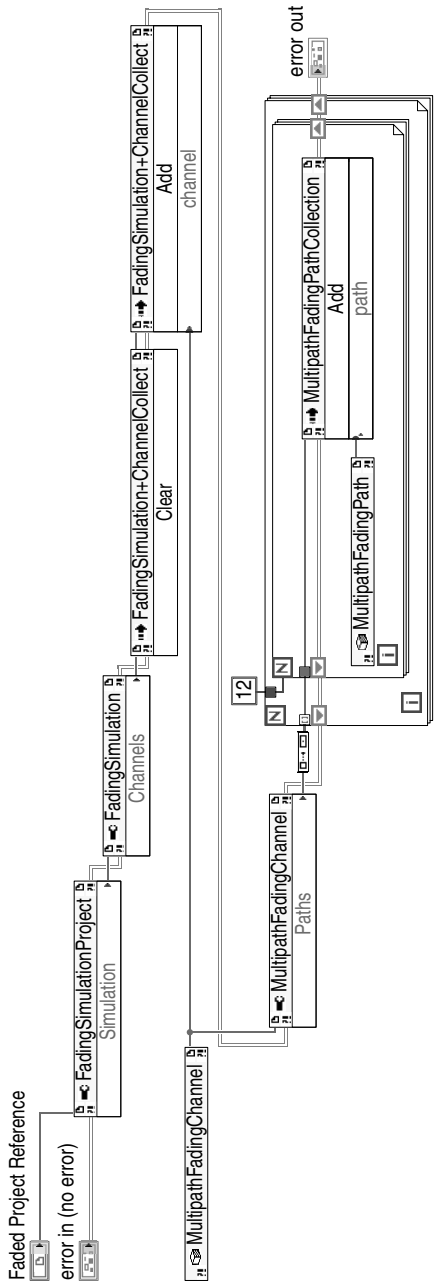


FIGURE 8.62

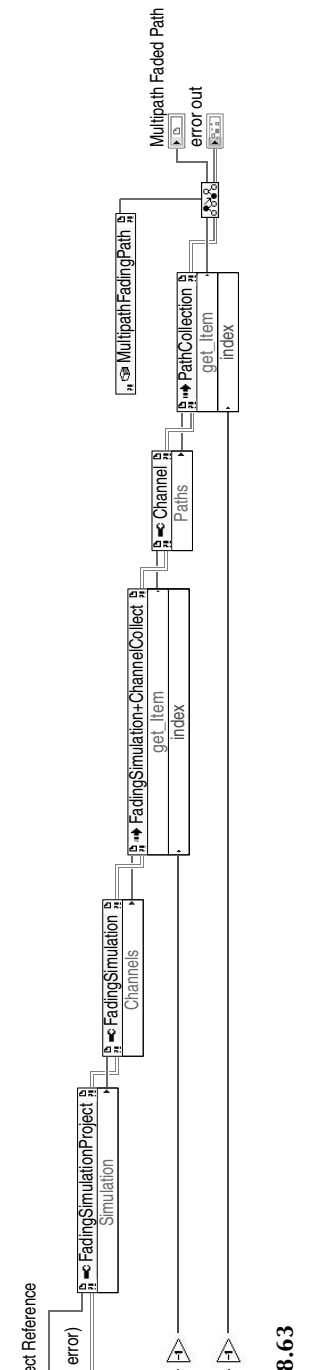


FIGURE 8.63

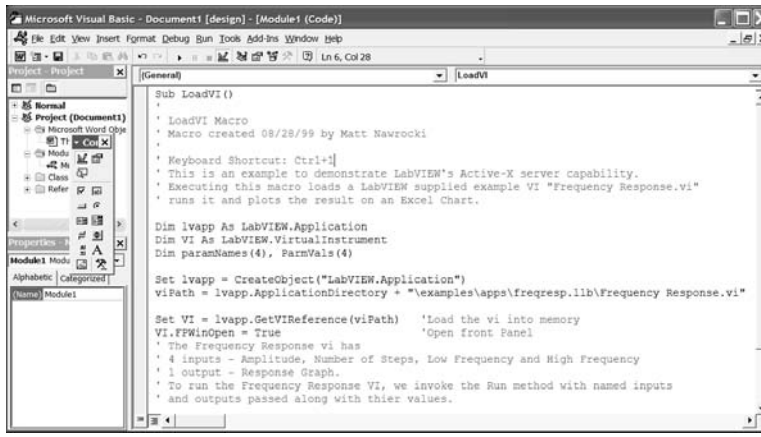


FIGURE 8.64

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 Microsoft Word. 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 8.64. 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 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 8.65.

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

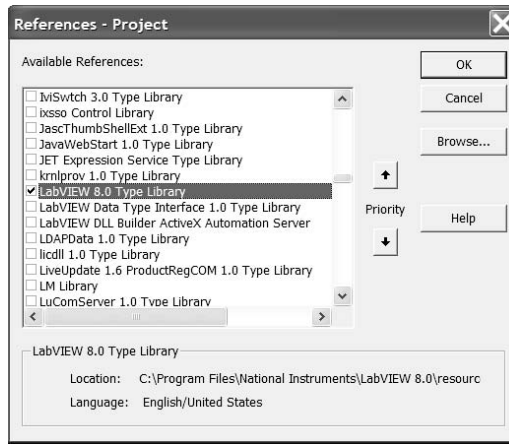


FIGURE 8.65

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. 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. Because 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 online reference. There is complete documentation there for interfacing LabVIEW to the rest of the COM-capable community.

8.9.16 Understanding ActiveX Error Codes

The ActiveX and the COM subsystems have their own sets of error codes. Virtually every ActiveX function has a return type that LabVIEW programmers cannot see. In C and C++, functions can have only 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 report problems only 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 multithreading 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 occur only 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.

At another time, 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.

8.9.17 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 lists only 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 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 see only 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.

BIBLIOGRAPHY

.Net Framework Essentials, 3rd ed. Thuan Thai and Hoang Q. Lam. O'Reilly and Associates Inc., Sebastopol, 2003, ISBN 0596005059.

Understanding ActiveX and OLE: A guide for developers and managers. David Chapell. Microsoft Press, Redmond, 1996, ISBN 1572312165.

G Programming Reference, National Instruments, Austin, TX, 1999.

9 Multithreading in LabVIEW

This chapter discusses using multithreading to improve LabVIEW applications' performance. Multithreading is an advanced programming topic, and its effective use requires the programmer to possess a fundamental understanding of this technology. LabVIEW provides two significant advantages to the programmer when working with multitasking and multithreading. The first advantage is the complete abstraction of the threads themselves. LabVIEW programmers never create, destroy, or synchronize threads. The second advantage is the data flow model used by LabVIEW. This model provides G a distinct advantage over its textual language counterparts because it simplifies a programmer's perception of multitasking. The fundamental concept of multitasking can be difficult to visualize with text-based languages.

Multithreading, as a technology, has been around for quite awhile. Windows 95 was certainly not the first operating system to deploy multithreading, but it was a key driver to get this technology widely deployed.

Multithreading added a new dimension to software engineering. Applications can perform multiple tasks somewhat simultaneously. A classic example of an application that has added multithreading is a word processor such as Microsoft Word or Open Office. Both applications use multithreading to perform spell-checking while Word also offers background grammar validation. The threads added to perform this task allow the application to perform these tasks while the user is typing. Now archaic versions, such as Word 6.0 for Windows 3.1, or modern versions, such as the word processors commonly found on PDAs, cannot do this because they run only one task at a time; a user would have to stop typing and select Check Spelling. The first six sections of this chapter provide the basic knowledge of multithreading. This discussion focuses on definitions, multitasking mechanics, multithreading specific problems, and information on various thread-capable operating systems. Once the basic mechanics of multithreading have been explained, we will explore a new variant of this: Hyper-Threading (HT) technology.

A brief section on multithreading myths is presented. The impact of multithreading on applications is misunderstood by a number of programmers. Section 9.6 explains precisely what the benefits of multithreading are. Many readers will be surprised to learn that multithreading does little to increase the speed of an application. Multithreading does provide the illusion that sections of an application run faster.

The last three sections of this chapter are devoted to the effective use of multithreading in LabVIEW. A strategy to estimate the maximum number of useful threads

will be presented. The focal point of this section is using subroutine VIs to maximize application performance. The use of threads adds a new dimension of benefits to both subroutine VIs and DLLs.

9.1 MULTITHREADING TERMINOLOGY

The following terminology will be used throughout this chapter. Programmers who require additional information on any of these topics should consult the chapter bibliography.

9.1.1 WIN32

Win32 is an Application Programming Interface (API) that is used by Microsoft's 32-bit operating systems: Windows XP home and office, Windows 2000, Windows NT, Windows 98, Windows 95, and Windows NT. The common API for programming also makes the operating systems look comparable to the programmer. Win32 replaced the Win16 API used in Windows 3.1, and in turn is being replaced by the Win64 API.

Windows XP and NT are designed to perform differently than Windows 95, 98, and ME. The primary focus of the Widows 9x line was to be as backward-compatible as possible with Windows 3.1. The Windows NT–derived product line, which includes XP, is designed to be as stable as possible. The differences in behavior of Windows NT line and Windows 9x are usually not obvious to most users. For example, many users do not know that Windows 9x will stop preemptive multithreading for certain Windows 3.1 applications whereas Windows XP will not.

9.1.2 UNIX

UNIX is an operating system that was conceived by AT&T Bell Labs. Like Win32, UNIX is a standard, but the Open Systems Foundation maintains the UNIX standard. Unlike Win32, UNIX is supported by a number of vendors who write and maintain operating systems to this standard. The most popular are Sun Microsystems' Solaris, IBM's AIX, Hewlett Packard's HP-UX, and the various Linux distributions.

Threads are specified in UNIX with the Portable Operating System Interface (POSIX) threads standard (pthread). At a low-level programming interface, pthreads are different than Win32 threads. This does not impact the LabVIEW programmer. Fundamentally, pthreads and Win32 threads operate in the same fashion. Their application-programming interfaces (API) are different, but conceptually, threads are threads.

9.1.3 MULTITASKING

Multitasking simply means to coordinate multiple tasks, which is a familiar concept to the LabVIEW programmer. As a dataflow-based language, LabVIEW has always supported multitasking — even the first edition of LabVIEW which ran on a single-threaded Macintosh OS. Operating systems, such as Windows 3.1 and MacOS, multitask operations such as multiple applications. The act of coordinating multiple

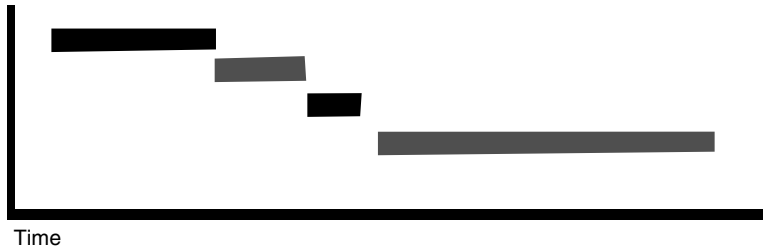


FIGURE 9.1

tasks should not be confused with multithreading; multithreading is fundamentally different, and this section will help to explain why.

The best example of multitasking is you. At work, you have a priority list of tasks that need to get accomplished. You will work on tasks one at a time. You may accomplish only some parts of a task, but not all before working briefly on another task. This is simple multitasking; you work on one thing for a while and then work on another topic. The process of switching between tasks is not multithreading; it is simply multitasking. The act of deciding how long to work on a task before working on a different one is your scheduling algorithm. This is the principle that Windows 3.1, MacOS, and some real-time operating systems were built on. In "the bad old days," the applications in Windows 3.1 and MacOS would decide how long they would run before surrendering the CPU to another process. These environments often had failures when an application would never decide to return the CPU to the system. Figure 9.1 demonstrates time utilization of a CPU when cooperative multitasking is used.

A simple demonstration of multitasking in LabVIEW is independent While loops. It is important for the reader to clearly understand that multitasking has always been available, and multithreading does not add or subtract from LabVIEW's ability to multitask operations. One of the key questions this chapter hopes to answer for the LabVIEW programmer is when multithreading will be of benefit.

9.1.3.1 Preemptive Multithreading

Taking the previous example of multitasking, you at work, we will explain the fundamental concept of multithreading. Imagine that your hands were capable of independently working. Your right hand could be typing a status report while the left dialed a phone number. Once the left hand completed dialing the number, it began to solder components on a circuit board. If you were capable of talking on the phone, typing your status report, and soldering components at the same time you would be multithreading. Your body is effectively a process, and your hands and mouth are threads of execution. They belong to the same process, but are functioning completely independent of each other.

This is fundamentally what multithreading is doing in a computer program. Each thread has a task it works on regardless of what the rest of the program is doing. This has been a difficult concept for many programmers to grasp. When programming with text-based languages such as C/C++, programmers associate lines of code

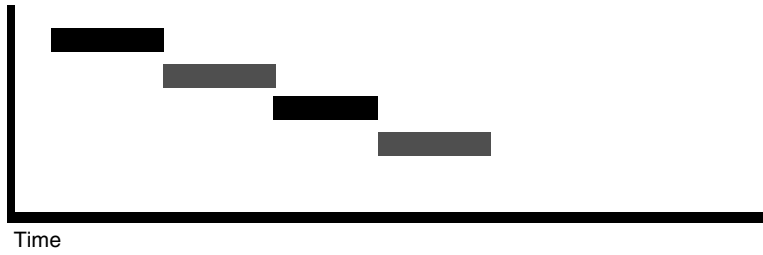


FIGURE 9.2

as operating sequentially. The concept that is difficult to grasp is that threads behave as their own little program running inside a larger one. LabVIEW's graphical code allows programmers to visualize execution paths much easier.

Preemptive multithreading uses CPU hardware and an operating system capable of supporting threads of execution. Preemption occurs at the hardware level; a hardware timer interrupt occurs and the CPU takes a thread of execution off the CPU and brings in another one. A lot of interesting things happen with CPUs that support multithreading operating systems. First, each program has a memory map. This memory map is unique to each application. The memory maps are translated into real memory addresses by hardware in the CPU. When preemption occurs, the timer in the CPU informs the CPU to change maps to the operating system's management. The operating system will then determine which thread is next to run and inform the CPU to load that thread's process memory map. Since the first edition of this book, all operating systems that LabVIEW supports are now capable of supporting multithreading. The act of scheduling threads and processes will be explained in Section 9.2. Figure 9.2 shows the timelines for multiple processes.

9.1.4 Kernel Objects

Kernel objects are small blocks of memory, often C structures, that are owned by the operating system. They are created at the request of programs and are used to protect sections of memory. Later in this chapter the need for protection will be clearly explained, but a short definition of this term is provided now.

9.1.5 Thread

Before we begin describing a thread, a few terms used for programs must be quickly defined. A program that has one thread of execution is a single-threaded program and must have a call stack. The call stack retains items like variables and what the next instruction is to be executed. C programmers will quickly associate variables placed on the stack as local variables. In some operating systems, the program will also have a copy of CPU registers. CPU registers are special memory locations inside the CPU. The advantage of using CPU registers is that the CPU can use these memory locations significantly faster than standard memory locations. The disadvantage is that there are relatively few memory locations in the registers. Each thread has its own call stack and copy of CPU registers.

Effectively, a thread of execution is a miniature program running in a large, shared memory space. Threads are the smallest units that may be scheduled time for execution on the CPU and possess a call stack and set of CPU registers. The call stack is a first-in-first-out (FIFO) stack that is used to contain things like function calls and temporary variables. A thread is aware of only its own call stack. The registers are loaded into the CPU when the thread starts its execution cycle, and pulled out and loaded back into memory when the thread completes its execution time.

9.1.6 Process

The exact definition of a process depends on the operating system, but a basic definition includes a block of memory and a thread of execution. When a process is started by the operating system, it is assigned a region of memory to operate in and has a list of instructions to execute. The list of instructions to begin processing is the "thread of execution." All applications begin with a single thread and are allowed to create additional threads during execution.

The process's memory does not correlate directly to physical memory. For example, a Win32 process is defined as four gigabytes of linear address space and at least one thread of execution. The average computer has significantly less memory. Banished are the bad old days of DOS programming where you had to worry about concepts such as conventional memory, extended memory, and high memory. These memory types still exist, but are managed by the operating system. The region created for a process is a flat, or linear, range of memory addresses. This is referred to as a memory map, or protected memory. The operating system is responsible for mapping the addresses the program has into the physical memory. The concept behind protected memory is that a process cannot access memory of other processes because it has no idea where other memory is. The operating system and CPU switch memory in and out of physical memory and hard disk space. Memory mapped to the hard disk is referred to as "virtual memory" in Windows and "swap space" in UNIX.

A process also has security information. This information identifies what the process has authorization to do in the system. Security is used in Windows XP professional and UNIX, but is generally not used in Windows XP Home or older Win32 operating systems such as Windows ME or 98.

9.1.7 Application

An application (or program) is a collection of processes. With concepts like Distributed Computing Environments (DCE) and the .NET environment, clustered applications, and even basic database applications, are not required to execute in one process, or even on one computer. With the lower cost of computers and faster network speeds, distributed computing is feasible and desirable in many applications. Applications that use processes on multiple machines are "distributed applications."

As an example of a distributed application, consider LabVIEW. With .NET support and VI Server functionality, VIs can control other VIs that are not resident and executing on the same computer. .NET is discussed in Chapter 8.

9.1.8 Priority

Every process and thread has a priority associated with it. The priority of the process or thread determines how important it is when compared to other processes or threads. Priorities are integer numbers, and the higher the number, the more important the process. Process priorities are relative to other processes while thread priorities are relative only to other threads in the same process. In other words, two running programs such as a LabVIEW executable running at priority 16 is less important than an operating system task operating a priority level 32. LabVIEW 8 by default has four threads. If you were to set one of these thread's personal thread priorities to 32 it would outrank the other LabVIEW process threads, but since the LabVIEW process still is out ranked by the operating system task the high priority LabVIEW thread will still be out ranked by all threads in the operating system process. LabVIEW programmers have access to priority levels used by LabVIEW. Configuring LabVIEW's thread usage will be discussed in Section 9.7.4.

9.1.8.1 How Operating Systems Determine which Threads

Both Win32 and POSIX have 32-integer values that are used to identify the priority of a process. The implementation of priority and scheduling is different between Win32 and pthreads. Additional information on both specifications appears in Sections 9.3 and 9.4.

9.1.9 Security

Windows XP and UNIX systems have security attributes that need to be verified by the operating system. Threads may operate only within the security restrictions the system places on them. When using .NET objects security permissions can become an issue. The default security attributes associated with an application are the level of permissions the user who started the application has. System security can limit access to files, hardware devices, and network components.

Windows .NET support adds additional security in the form of "managed code." It needs to be understood that .NET overlays onto the core Windows operating system. If the operating system does not support security features (Windows 9X and ME), then it does not fully support .NET's security features.

9.1.10 Thread Safe

Programmers often misunderstand the term "thread safe." The concept of thread-safe code implies that data access is atomic. "Atomic" means that the CPU will execute the entire instruction, regardless of what external events occur, such as interrupts. Assembly-level instructions require more than one clock cycle to execute, but are atomic. When writing higher-level code, such as LabVIEW VIs or C/C++ code, the concept of executing blocks of code in an atomic fashion is critical when multiple threads of execution are involved. Threads of execution are often required to have access to shared data and variables. Atomic access allows for threads to have complete access to data the next time they are scheduled.

The problem with preemption is that a thread is removed from the CPU after completion of its current machine instruction. Commands in C can be comprised of dozens of machine-level instructions. It is important to make sure data access is started and completed without interference from other threads. Threads are not informed by the operating system that they were preemptively removed from the CPU. Threads cannot know shared data was altered when it was removed from the CPU. It is also not possible to determine preemption occurred with code; it is a hardware operation that is hidden from threads. Several Kernel objects can be used to guarantee that data is not altered by another thread of execution.

Both UNIX Pthreads and Win32 threads support semaphores and mutexes. A Mutual Exclusion (mutex) object is a Kernel object that allows one thread to take possession of a data item. When a thread requires access to a data item that is protected by a mutex, it requests ownership from the operating system. If no other threads currently own the mutex, ownership is granted. When preemption occurs, the owning thread is still shifted off the CPU, but when another thread requests ownership of the mutex it will be blocked. A thread that takes possession of a mutex is required to release the mutex. The operating system will never force a thread to relinquish resources it has taken possession of. It is impossible for the operating system to determine if data protected by the mutex is in a transient state and would cause problems if another thread were given control of the data.

A semaphore is similar to a mutex, but ownership is permitted by a specified number of threads. An analogy of a semaphore is a crowded nightclub. If capacity of the club is limited to 500 people, and 600 people want to enter the club, 100 are forced to wait outside. The doorman is the semaphore, and restricts access to the first 500 people. When a person exits, another individual from outside is allowed to enter. A semaphore works the same way.

Mutexes and semaphores must be used in DLLs and code libraries if they are to be considered thread-safe. LabVIEW can be configured to call DLLs from the user interface subsystem, its primary thread, if it is unclear that the DLL is thread safe. A programmer should never assume that code is thread safe; this can lead to very difficult issues to resolve.

9.2 THREAD MECHANICS

All activities that threads perform are documented in an operating system's specification. The actual behavior of the threads is dependent on a vendor's implementation. In the case of Windows, there is only one vendor, Microsoft. On the other hand, Linux and UNIX have a number of vendors who implement the standard in slightly different ways. Providing detailed information on operating system-specific details for all the UNIX flavors is beyond the scope of this book.

Regardless of operating system, all threads have a few things in common. First, threads must be given time on the CPU to execute. The operating system scheduler determines which threads get time on the CPU. Second, all threads have a concept of state; the state of a thread determines its eligibility to be given time on the CPU.

9.2.1 Thread States

A thread can be in one of three states: active, blocked, or suspended. Active threads will be arranged according to their priority and allocated time on the CPU. An active thread may become blocked during its execution. In the event an executing thread becomes blocked, it will be moved into the inactive queue, which will be explained shortly.

Threads that are blocked are currently waiting on a resource from the operating system (a Kernel object or message). For example, when a thread tries to access the hard drive of the system, there will be a delay on the order of 10 ms for the hard drive to respond. The operating system blocks this thread because it is now waiting for a resource and would otherwise waste time on the CPU. When the hard drive triggers an interrupt, it informs the operating system it is ready. The operating system will signal the blocked thread and the thread will be put back into a run queue.

Suspended threads are "sleeping." For example, in C, using the Sleep statement will suspend the thread that executed the command. The operating system effectively treats blocked and suspended threads in the same fashion; they are not allowed time on the CPU. Both suspended and blocked threads are allowed to resume execution when they have the ability to, when they are signaled.

9.2.2 Scheduling Threads

The operation of a scheduling algorithm is not public knowledge for most operating systems, or at least a vendor's implementation of the scheduling algorithm. The basic operation of scheduling algorithms is detailed in this section.

The operating system will maintain two lists of threads. The first list contains the active threads and the second list contains blocked and suspended threads. The active thread list is time-ordered and weighted by the priority of the thread in the system. The highest priority thread will be allowed to run on the CPU for a specified amount of time, and will then be switched off the CPU. This is referred to as a "round robin" scheduling policy.

When there are multiple CPUs available to the system, the scheduler will determine which threads get to run on which CPU. Symmetric Multiprocessing (SMP) used in Windows XP Professional allows for threads of the same process to run on different CPUs. This is not always the case. A dual-CPU machine may have threads of different processes running on the pair of CPUs, which is determined by the scheduling algorithm. Some UNIX implementations allow only a process's threads on a single CPU.

The blocked/suspended queue is not time-ordered. It is impossible for the operating system to know when a signal will be generated to unblock a thread. The scheduling algorithm polls this list to determine if any threads have become available to execute. When a thread becomes unblocked and has higher priority than the currently running thread, it will be granted control of the CPU.

9.2.3 Context Switching

The process of changing which thread is executing on the CPU is called "context switching." Context switching between threads that are in the same process is

relatively fast. This is referred to as a "thread context switch." The CPU needs to offload the current thread's instruction pointer and its copy of the CPU registers into memory. The CPU will then load the next thread's information and begin executing the next thread. Since threads in the same process execute in the same protected memory, there is no need to remap physical memory into the memory map used by the process.

When context switching occurs between threads of different processes, it is called a "process context switch." There is a lot more work that is required in a process context switch. In addition to swapping out instruction pointers and CPU registers, the memory mapping must also be changed for the new thread.

As mentioned in Section 9.2.2, when a thread becomes signaled (eligible to run) and has a higher priority than the currently running thread, it will be given control of the CPU. This is an involuntary context switch. This is a potential problem for LabVIEW programmers. Section 9.5 will discuss multithreading problems such as starvation and priority inversion that can be caused by poorly-designed configurations of LabVIEW's thread pools. Configuring the threads that LabVIEW uses is discussed in Section 9.7.4.

9.3 WIN32 MULTITHREADING

The Win32 model expands the capabilities of the Win16 model. Threading and security are two of the features that were added to Windows. Win32 systems branched with the release of Windows NT 3.51. Windows 2000 and XP leveraged the threading model used in Windows NT. Windows NT and Windows 95 operate differently when threads are considered, as stated-modern Windows operating systems use the Windows NT model for thread handling. The API for these operating systems are the same. Windows 98 and ME ignore several attributes given to a thread. For example, Windows 98 and ME ignore security attributes. Windows 98 and ME were designed for home usage so thread/process security is not used in these operating systems. Issues involving security and permissions are usually not a problem in LabVIEW applications. Some issues may surface when using the VI server or .NET objects.

Windows XP will always operate using preemptive multithreading. The primary design goal of Windows NT branch was stability. Windows 95x platforms and 98 were designed with backward compatibility in mind. Windows 9x will run as cooperative multithreaded environments, similar to Windows 3.1. This is a consideration for LabVIEW programmers to remember when they anticipate a need to support users running legacy applications such as Microsoft Office Version 4 or interfacing with very old test applications. Older versions of Microsoft Word (Version 6.0) may be given cooperative multitasking support so they feel at home in a Windows 3.1 environment. Legacy DOS applications may behave the same way. Either way, when Windows 9x systems drop into a cooperative multitasking mode it will not inform the user that this decision was made. Performance degradation may be seen in Windows 9x environments when legacy applications are run. If users raise issues related to performance of LabVIEW applications on Windows 9x systems, ask if any other applications are being used concurrently with LabVIEW.

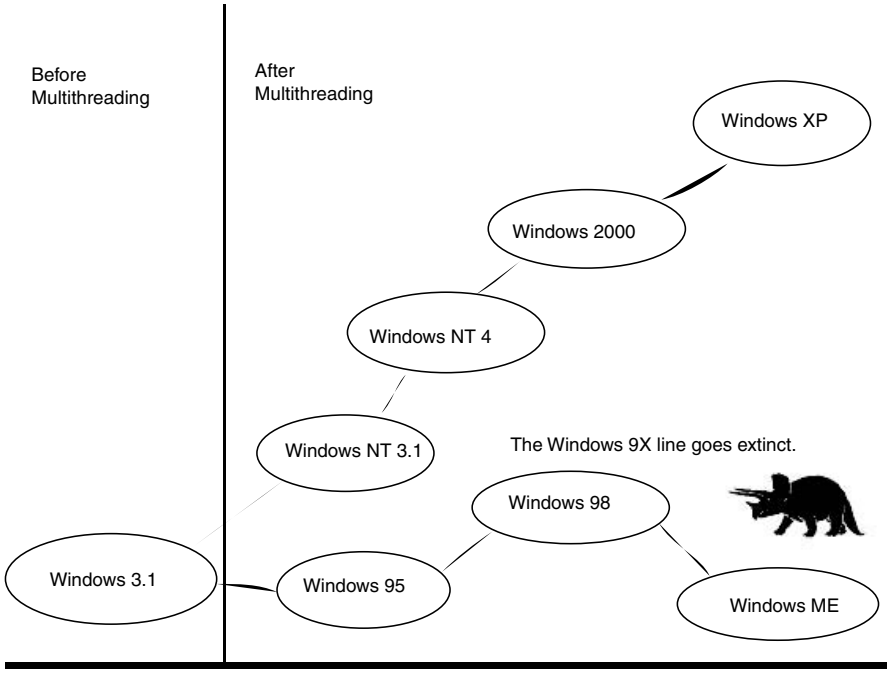


FIGURE 9.3

Another difference between the Windows 9x and NT branch is hardware accessibility. Windows 9x allows applications to access hardware directly. This allows drivers to have faster access to devices such as GPIB cards and DAQ boards. Windows NT's model for stability forbids direct hardware access. Drivers must be written specifically to act as operating system components.

A useful chart showing the progression of Microsoft's operating systems is shown in Figure 9.3 Windows operating systems did a "split" with Windows 3.1 and Windows NT 3.1. The Windows 95 branch has essentially died out. All operating systems since Windows ME have been on the Windows NT branch.

9.4 PTHREADS

Many topics on UNIX multithreading are well beyond the scope of this book. The chapter bibliography lists a number of sources that contain additional information. One of the difficulties in writing about UNIX is the number of vendors writing UNIX. The POSIX standard is intended to provide a uniform list of design requirements for vendors to support. This does not translate directly to uniform behavior of UNIX operating systems. Vendors write an operating system that conforms to their interpretation of the specification.

Priority and scheduling are different for Pthreads; Pthreads have defined scheduling policies: round robin; first-in, first-out; and others. The FIFO policy lets a thread execute until it completes its execution or becomes blocked. This policy is

multitasking by any other name, because there is no preemption involved. The round-robin policy is preemptive multithreading. Each thread is allowed to execute for a maximum amount of time, a unit referred to as a "quantum." The time of a quantum is defined by the vendor's implementation. The "other" policy has no formal definition in the POSIX standard. This is an option left up to individual vendors. Pthreads expand on a concept used in UNIX called "forking." A UNIX process may duplicate itself using a fork command. Many UNIX daemons such as Telnet use forking. Forking is not available to the Win32 programmer. A process that generates the fork is called the Parent process, while the process that is created as a result of the fork command is referred to as the Child process. The Child process is used to handle a specific task, and the Parent process typically does nothing but wait for another job request to arrive. This type of multitasking has been used for years in UNIX systems.

As an example of forking, consider a Telnet daemon. When a connection request is received, the process executes a fork. The Telnet daemon is replicated by the system and two copies of the daemon are now running in memory; the Parent process and Child process. The Parent continues to listen on the well-known Telnet port. The Child process takes over for the current Telnet connection and executes as an independent process. If another user requests another Telnet session, a second Child process will be spawned by another fork. The Parent process, the original Telnet daemon, will continue to listen to the well-known Telnet port. The two Child processes will handle the individual Telnet sessions that users requested.

Forking has both advantages and disadvantages. The major disadvantage to forking is that Parent and Child processes are independent processes, and both Parent and Child have their own protected memory space. The advantage to having independent memory spaces is robustness; if the Child process crashes, the Parent process will continue to execute. The disadvantage is that since Parent and Child are independent processes, the operating system must perform process context switches, and this requires additional overhead.

9.5 MULTITHREADING PROBLEMS

This section will outline problems that multithreading can cause. This is an important section and will be referenced in Section 9.6, "Multithreading Myths." Some of these problems do not occur in LabVIEW, but are included for completeness. Many of these problems can be difficult to diagnose because they occur at the operating system level. The OS does not report to applications that cause these problems. Some issues, such as race conditions and deadlock, cannot be observed by the operating system. The operating system will not be able to determine if this is normal operation or a problem for the code.

It is important to understand that any multitasking system can suffer from these problems, including LabVIEW. This section is intended to discuss these problems relative to LabVIEW's multithreading abilities. As a programmer, you can create race conditions, priority inversion, starvation, and deadlock in LabVIEW's VIs when working with VI priorities. We will identify which problems you can cause by poor configuration of LabVIEW's thread counts.

9.5.1 Race Conditions

Many LabVIEW programmers are familiar with the concept of a "race condition." Multithreading in general is susceptible to this problem. Fortunately, when writing LabVIEW code, a programmer will not create a race condition with LabVIEW's threads. A race condition in multithreaded code happens when one thread requires data that should have been modified by a previous thread. Additional information on LabVIEW race conditions can be found in the LabVIEW documentation or training course materials. LabVIEW execution systems are not susceptible to this problem. The dedicated folks at National Instruments built the threading model used by LabVIEW's execution engine to properly synchronize and protect data. LabVIEW programmers cannot cause thread-based race conditions; however, there is still plenty of room for LabVIEW programmers to create race conditions in their own code.

Thread-based race conditions can be extremely hazardous. If pointer variables are involved, crashes and exception errors are fairly likely. This type of problem can be extremely difficult to diagnose because it is never made clear to the programmer the order in which the threads were executing. Scheduling algorithms make no guarantees when threads get to operate.

9.5.2 Priority Inversion

A problem occurs when two threads of different priority require a resource. If the lower-priority thread acquires the resource, the higher-priority process is locked out from using the resource until it is released. Effectively, the higher-priority process has its priority reduced because it must now wait for the lower-priority process to finish. This type of blocking is referred to as Priority Inversion. The resource could be in the application, or the resource could be external to the application, such as accessing a shared file on the hard drive. Internal resources include things like variables. External resources include accessing the hard drive, waiting on .NET components, and waiting on Kernel-level operating system components such as mutexes and semaphores.

Priority inversion will degrade an application's performance because high-priority threads do not execute as such. However, the program might still execute properly. Inversion is a problem that can be caused by a LabVIEW programmer who errantly alters the priority level of LabVIEW's thread pools. Priority levels of "Normal" should be used, and this will prevent priority inversion problems. When threads have no active work to do, they will be blocked.

Most LabVIEW applications should not require modification of priority levels of threads. Errant priority levels can cause a number of problems for a LabVIEW application. An example of when a programmer would consider adjusting the priority level of a subsystem is a high-speed data acquisition program. If the DAQ subsystem required a majority of the CPU time, then a programmer may need to raise priority levels for the DAQ subsystem. If a common queue were used to store data brought in from the DAQ card, priority inversion would occur. VIs performing numerical processing or data display that are resident in the user subsystem will execute with lower priority than the DAQ threads. This becomes a problem when the user interface

threads have access to the queue and spend a lot of time waiting for permission to execute. The DAQ threads trying to put data into the queue become blocked until lower-priority interface threads complete. The lesson to learn here is important: when priority inversion occurs, the high-priority threads end up suffering. This type of problem can seriously impact application performance and accuracy. If the DAQ card's buffer overflows because the DAQ subsystem was blocked, application accuracy would become questionable.

9.5.3 Starvation

Starvation is essentially the opposite of priority inversion. If access to a resource that two threads need is for very short periods of time, then the higher-priority thread will almost always acquire it before the lower priority thread gets the opportunity to do so. This happens because a higher-priority thread receives more execution time. Statistically, it will get the resource far more often than the lower-priority thread.

Like, priority inversion, "starvation" is resolved differently by the Windows NT and 9x branches — recall Windows XP is on the NT branch. Again, like priority inversion, a LabVIEW programmer can cause this multithreading problem. We will discuss prevention of both priority inversion and starvation issues later in this chapter.

If Windows will actively seek to resolve starvation and priority inversion, then why bother to prevent them from happening? The reason you do not want to cause either problem is that both reduce the efficiency of an application. It is poor programming practice to design an application that requires the operating system to resolve its deficiencies.

If there is a valid reason for a thread to execute at a higher priority than others, then the program design should make sure that its execution time is not limited by priority inversion. Lower-priority threads can suffer from starvation. This is highly undesirable because execution time of the higher-priority threads will become limited while Windows allows the lower threads to catch up. A balanced design in thread priorities is required. In multithreaded code, it is often best just to leave all threads at the same priority.

9.5.4 Deadlocking

Deadlock is the most difficult multithreading problem that is encountered. Deadlock can occur only when two or more threads are using several resources. For example, there are two threads, A and B. There are also two resources, C and D. If both threads need to take ownership of both resources to accomplish their task, deadlocking occurs when each thread has one resource and is waiting for the other. Thread A acquires resource C, and thread B acquires resource D. Both threads are blocked until they acquire the other resource.

The bad news as far as deadlocking is concerned: Windows and Linux have no mechanism to resolve this type of problem. The operating system will never force a thread to release its resources. Fortunately, deadlocking is highly unlikely to be caused by a LabVIEW programmer. This is a thread-level problem and would be caused by the execution engine of LabVIEW. Once again, the dedicated folks at

National Instruments have thoroughly tested LabVIEW's engine to verify that this problem does not exist. This eliminates the possibility that you, the programmer, can cause this problem.

9.5.5 Operating System Solutions

Operating systems try to compensate for starvation and priority inversion. Requiring the operating system to compensate is poor programming practice, but here is how UNIX and Win32 try to resolve them.

Priority inversion is resolved differently by Windows 9x and NT branches. Windows NT-derived systems (Windows XP) will add a random number to the priority of every thread when it orders the active queue. This obviously is not a complete solution, but a well-structured program does not require the operating system to address thread problems.

Windows 9x will actually increase the priority of the entire process. Increasing the priority of the process gives the entire application more of the CPU time. This is not as effective a solution as the one used by NT. The reason for this type of handling in Windows 95 is for backward compatibility. Legacy Win16 programs are aware of only one thread of execution. Elevating the entire process's priority makes Win16 applications feel at home in a Windows 3.1 environment. You may be very hard pressed to find systems out there still using Win16 applications, but the support for these types of applications still exist in the Win9x operating systems.

Since the first edition of this book, the entire Win9x line of operating systems has largely fallen out of use and Microsoft has ended support for the operating systems in this line.

9.6 MULTITHREADING MYTHS

This section discusses some common myths about multithreading. We have heard many of these myths from other LabVIEW programmers. Multithreading is one of the exciting new additions to the language that first appeared in LabVIEW 5.0; unfortunately, it is also one of the most dangerous. The following myths can lead to performance degradation for either a LabVIEW application or the entire operating system. The single biggest myth surrounding multithreading is that it makes applications run faster. This is entirely false, and the case is outlined below.

9.6.1 The More Threads, the Merrier

The first myth that needs to be addressed is "More threads, better performance." This is just not in line with reality; application speed is not a function of the number of running threads. When writing code in languages like C++, this is an extremely dangerous position to take. Having a thread of execution for every action is more likely to slow an application down than speed it up. If many of the threads are kept suspended or blocked, then the program is more likely to use memory inefficiently. Either way, with languages like C++ the programmer has a lot of room to cause significant problems.

LabVIEW abstracts the threading model from the programmer. This threading model is a double-edged sword. In cases like the number of threads available, LabVIEW will not always use every thread it has available, and the programmer will just waste memory. When applications are running on smaller computers, such as laptops where memory might be scarce, this could be a problem.

The rule of thumb for how many threads to have is rather vague: do not use too many. It is often better to use fewer threads. A large number of threads will introduce a lot of overhead for the operating system to track, and performance degradation will eventually set in. If your computer has one CPU, then no matter what threading model you use, only one thread will run at a time. We'll go into customizing the LabVIEW threading model later in this chapter. We will give guidelines as to how many threads and executions systems programs might want to have.

9.6.2 More Threads, More Speed

This myth is not as dangerous as "The more threads the merrier" but still needs to be brought back to reality. The basic problem here is that threads make sections of a program appear to run at the same time. This illusion is not executing the program faster. When the user interface is running in its own thread of execution, the application's GUI will respond more fluidly. Other tasks will have to stop running when the user interface is updating. When high performance is required, a nicely updated GUI will degrade the speed that other subsystems operate.

The only true way to have a program run faster with multiple threads is to have more than one CPU in the computer. This is not the case for most computers, and, therefore, threads will not always boost application performance. Most modern Intel-based machines for both Windows and Linux have Hyper-Threading capable processors. We will discuss Hyper-Threading in the next section. Oddly enough, Hyper-Threading does not always improve the performance of a multithreaded application. Windows 9x and XP home do not support Symmetric Multiprocessing, and will make use of only one CPU in the system.

Later in this chapter, we will show how performance gains can be made without making significant changes to the thread configuration. As a rule of thumb, threads are always a difficult business. Whenever possible, try to tweak performance without working with threads. We will explain the mechanics of multithreading in the next sections, and it will become clear to the reader that multithreading is not a "silver bullet" for slow applications.

9.6.3 Makes Applications More Robust

There is nothing mentioned in any specification regarding threads that lends stability to an application. If anything, writing multithreaded code from a low level is far more difficult than writing single-threaded code. This is not a concern for the LabVIEW programmer because the dedicated folks at National Instruments wrote the low-level details for us. Writing thread-safe code requires detailed design and an intimate understanding of the data structures of a program. The presence of multiple threads does not add to the stability of an application. When a single thread

in a multithreaded application throws an exception, or encounters a severe error, it will crash the entire process. In a distributed application, it could potentially tear down the entire application. The only way to ensure the stability of a multithreaded application is a significant amount of testing. National Instruments has done this testing, so there is no need to spend hours doing so to verify your threading models.

9.6.4 Conclusion on Myths

Multithreading gives new abilities to LabVIEW and other programming languages, but there is always a price to be paid for gains. It is important to clearly understand what multithreading does and does not provide. Performance gains will not be realized with threads running on a single-processor system. Large numbers of threads will slow an application down because of the overhead the system must support for the threads to run. Applications' graphical interfaces will respond more fluidly because the graphics subsystems' threads will get periodic time to execute. Intensive computational routines will not block other sections of code from executing.

9.7 HYPER-THREADING

Before we can begin a discussion of how Hyper-Threading works, we will need to review the "instruction pipeline" that a processor uses. Modern processors use a variety of buffers, queues, caches of memory, architecture states, stack points, and registers. These items tend to place limits on the performance of a given processor. For example, cache memory is always a performance tradeoff. It is always desirable for a larger memory cache near the processor, but the cache's speed performance is inversely related to its size; make the cache small and it runs faster; make it larger and it runs more slowly.

Intel has added a new technology to their processors; marketing literature refers to it as Hyper-Threading. Hyper-Threading's technical name is *simultaneous multi-threading*. Hyper-Threading adds a second set of CPU resources to a processor to make it appear in most respects to have two processors.

Hyper-Threading is not the same as Symmetric Multiprocessing. SMP is two completely independent processors; a Hyper-Threading processor has two cores that have to share some key resources of the processor. In order to take advantage of Hyper-Threading, the processor, BIOS, chip set, and operating system all need to be capable of supporting Hyper-Threading. Windows XP was the first Microsoft operating system with Hyper-Threading support.

Hyper-Threading implements the concepts of logical and physical processors. An HT processor is one physical processor that implements two logical processors. The operating system will present the logical processors as two processors to the system. If you look at Window's task manager on an HT-enabled machine, the performance tab will show two CPUs. The two logical processors are not two completely independent processors — elements of a CPU core are either replicated, shared, or partitioned. The HT processor will start up and shut down the second logical core as needed to improve performance of the entire chip.

Items the CPU core has to share between the two cores are the cache memory and the out of order execution engine. The out of order execution engine is available

on P6 or later Intel processors. This engine reviews running code for dependencies. If two instructions do not conflict on resources running on the CPU, it will run them at the same time in parallel. The instructions and their results have to be re-sorted into order after the parallel operation. The memory cache is a set of high speed memory near the processor that allows for the CPU to have quick access to relevant memory. These two items are shared by the CPU cores in HT processors. As the two cores have to share, the performance of these segments can be reduced per thread running.

The buffers to reorder instructions from the out of order execution engine and queues in the core are partitioned. There is one set of each, and parts are allocated specifically to one of the cores. When the performance of a single core would be greatly improved by allocating all these resources to the single core, the HT processor can shut down the second virtual processor which partitions all of the queues and recording buffers to the single core.

Each core will have its own instruction pointer and thread state. In the event the second core is shut down, this hardware will not be used by the processor.

A Hyper-Threading microprocessor has a common process context between the cores. In other words, a HT-enabled system can run two threads belonging to the same process together. Multiple processes cannot be run simultaneously because every process has an independent memory mapping — the hardware to do the map to real memory translation is shared by both cores. In general, there will always be operating system specific tasks running. Of importance to the LabVIEW programmer will be processes that handle I/O such as Internet-related traffic. Windows XP and Linux do not permit direct hardware access. Anytime I/O is being used, it can be assumed that a driver running in a different mode of operation is in use.

LabVIEW 8 by default uses four threads of execution. Hyper-Threading will not always run two threads at a time. If three of the threads are idle or blocked, the processor will shut down the second core and provide all processor resources to the active thread. When two threads are running, they will run a bit slower at the same time compared to the execution speed of one thread running with all CPU resources. It is possible to combine SMP and HT. In the event you have a dual processor machine with two HT-enabled processors the system will operate with four logical processors.

The average thread of execution uses roughly 35% of a processor's capabilities. Hyper-Threading adds enough logic circuitry to the processor to allow for a second thread to run and take advantage of the remaining resources. The relatively small increase in chip hardware for generally better performance makes Hyper-Threading a very useful technology. HT-enabled processors do not significantly increase the power consumption of a processor, which gives a significant improvement in performance relative to power consumption. In environments with a lot of hardware running, there are realizable savings in electrical usage for the machines and the HVAC systems that support them.

9.8 MULTITHREADED LABVIEW

Fundamentally, the dataflow operation of LabVIEW is not impacted by multithreading. The real differences are abstracted from the programmer. In this section we

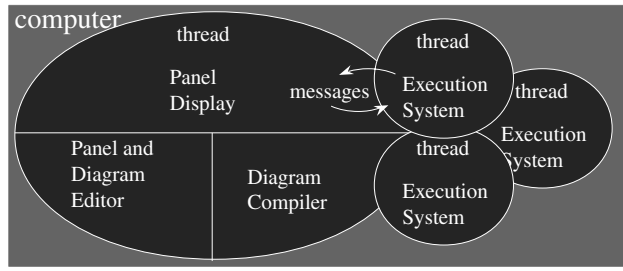


FIGURE 9.4

discuss the architecture of multithreaded LabVIEW, the main run queue, and how to configure the thread systems. Understanding how threads interact with the LabVIEW subsystems will allow a programmer to use threading effectively. Threads in one subsystem may access VIs you thought were operating in another subsystem. The topic is somewhat abstract, and this section intends to clarify the interactions of VIs and threads. Once the reader understands the subsystem architecture and main run queue, an introduction to thread configuration will be presented.

9.8.1 Execution Subsystems

The various activities that LabVIEW performs are handled by six subsystems. The LabVIEW subsystems are User, Standard, I/O, DAQ, Other 1, and Other 2. The original design of LabVIEW 5.0 used these subsystems to perform tasks related to the system's name. This rigid partitioning did not make it into the release version of LabVIEW 5.0, and tasks could run in any LabVIEW subsystem. This subsystem architecture still exists in LabVIEW 8. Figure 9.4 depicts LabVIEW broken up into its constituent subsystems.

Each subsystem has a pool of threads and task queue associated with it. LabVIEW also maintains a main run queue. The run queue stores a priority-sorted list of tasks that are assigned to the threads in the subsystem. A LabVIEW subsystem has an "array" of threads and priorities. The maximum number of threads that can be created for a subsystem is 40; this is the maximum of 8 threads per priority and 5 priority levels. Section 9.8.4 discusses thread configuration for subsystems. Figure 9.5 shows a subsystem and its array of threads.

The User subsystem is the only subsystem that is required for LabVIEW to run, because all other subsystems are optional to running LabVIEW. Configuring thread counts and priorities for subsystems is covered in 9.8.4. The User subsystem maintains the user interface, compiles VIs, and holds the primary thread of execution for LabVIEW. When a DLL or code fragment with questionable thread safety is run, the User subsystem is where it should always be called. LabVIEW can be configured to run in as a single threaded application. The single thread runs and it rides on the User subsystem.

The Standard subsystem was intended to be the default subsystem for LabVIEW executable code. If you are interested in keeping dedicated execution time to the user interface, assign the main level VIs to this subsystem. This will guarantee that

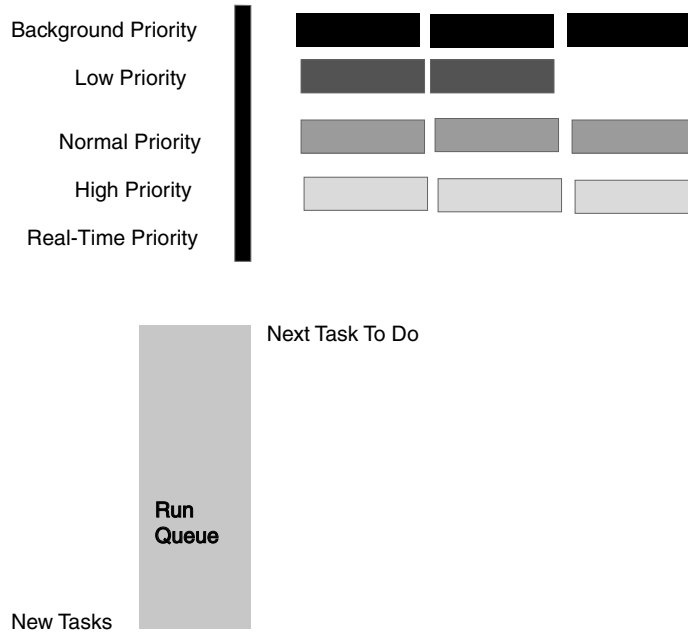


FIGURE 9.5

the User subsystem threads have plenty of time to keep the display updated. Like the User subsystem, the Standard subsystem can be configured to run with an array of threads.

The DAQ subsystem was originally intended to run data acquisition-specific tasks. It is currently available to any VI. The I/O subsystem was intended for VXI, GPIB, Serial, and IP communications (TCP and UDP). This design is interesting when the programmer considers using the VISA communication suite. VISA is a communications subsystem, and a dedicated thread pool is certainly a good idea. Remember that having dedicated threads guarantees some amount of execution time. On single CPU systems without Hyper-Threading, there is still a "borrow from Peter to pay Paul" issue, but communications is fundamental to many LabVIEW applications and justification for guaranteed execution time is sometimes appropriate. The priority levels of a subsystem's threads relative to the other subsystem threads would serve as a rough gauge of the amount of execution time available. Other 1 and Other 2 were intended for user-specified subsystems. Again, these two subsystems can be used for any purpose desired. Most applications do not need to assign VIs to specific subsystems. A simple litmus test to decide if a VI should be dedicated to a subsystem is to write a one-paragraph description of why the VI should only be executed in a specific subsystem. This is a simple Description Of Logic (DOL) statement that should be included in any application design. Description of Logic was discussed in Chapter 4. If the DOL cannot describe what the benefits of the assignment are, then the assignment is not justified. Valid reasons for dedicating a set of VIs to a specific subsystem include the need to guarantee some amount of execution time.

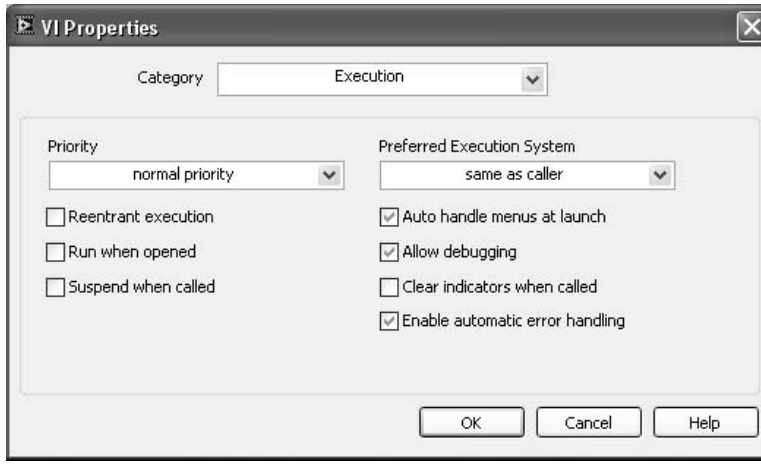


FIGURE 9.6

If a programmer decides to modularize the execution of an application, then assignment of a VI to a subsystem can be done in VI Setup as shown in Figure 9.6.

When a new VI is created, its default priority is normal and the system is "same as caller." If all VIs in a call chain are listed as run under "same as caller," then any of the subsystems could potentially call the VI. The default thread configuration for LabVIEW is to have four threads per subsystem with normal priority. The subsystem that calls a VI will be highly variable; during execution it is impossible to determine which threads were selected by the scheduler to execute. When a VI is assigned to a particular subsystem, only threads belonging to the specified subsystem will execute it.

Now that a simple definition of the LabVIEW subsystems has been presented, let's consider threads. Subsystem threads execute in a round-robin list and are scheduled by the operating system. When a VI is listed to execute in a specific subsystem, only threads assigned to that subsystem can execute it. As an example, a VI, test_other1.vi is permitted to be executed only by the Other 1 subsystem. When test2_other2.vi is executed and told to call test_other1.vi, the thread that is executing test2_other2.vi will block. The data flow at the call is blocked until a thread from the Other 1 subsystem is available to execute it. This is up to the scheduler of the operating system, and also one of the points where errant priorities can cause priority inversion or starvation. LabVIEW cannot directly switch to a thread in Other 1 to execute the thread. Only the operating system can decide which thread gets to execute when. Other 1 threads will not be considered special to the operating system, and they must wait in the thread queue until it is their turn to execute.

When assigning VIs to particular subsystems, use caution when assigning thread priorities. If VIs are called by two subsystems, there should not be large differences in the priorities of the threads belonging to these subsystems. An example is if the subsystem Other 1 has 2 threads at "Above Normal" priority and Other 2 has a "Background" priority thread. If a VI called in subsystem Other 1 calls a subVI in Other 2, not only does the thread in Other 1 block, it has to wait for a background

priority thread in Other 2 to get scheduled time to execute! This is a simple example of priority inversion that can be caused by a LabVIEW programmer. If Other 1 and Other 2 were both executing with "Normal" priority, the scheduling algorithm would have mixed the scheduling of the threads and no priority inversion would occur. The thread in Other 1 would have needed to wait on the thread context switches, but those delays are relatively minor.

Another threading problem that can be caused by errant priority settings is starvation. Consider the following case: VIs other1.vi, other2.vi, and io.vi. The Other 1 VI is listed with "time-critical" priority level, Other 2 VI is listed with "background" priority, and io.vi is listed "same as caller." Same as caller allows a VI to execute in any subsystem with the same priority thread that executed the calling VI. Both other1.vi and other2.vi need to call io.vi. Since other1.vi is running in a subsystem with a time-critical priority thread, it is going to get a significant amount of execution time compared with other2.vi. Access to io.vi will be granted to other1.vi far more often than other2.vi. Other2.vi will become starved because it does not get enough access to io.vi. Scheduling algorithms are notoriously unforgiving when they schedule threads. If a thread is available to execute, it will get put in the active list based entirely on its priority. If a thread with low priority is always available, it will still make the active list, but will always be near the bottom.

9.8.2 The Run Queue

LabVIEW maintains several run queues consisting of a main run queue and a run queue for each subsystem. A run queue is simply a priority-ordered list of tasks that are executed. When a VI is executed, the LabVIEW execution engine determines which elements in the block diagram have the needed inputs to be executed. The engine then orders inputs by priority into the run queues. The run queue is not strictly a first-in first-out (FIFO) stack. VIs have priorities associated with them (the default priority is "normal"). After execution of each element, the run queue is updated to reflect elements (subVIs or built-in LabVIEW functions, such as addition, subtraction, or string concatenation) that still need to be executed. It is possible for VIs to take precedence over other VIs because they have higher priority. Wildly changing VI priorities will likely result in performance issues with LabVIEW. One key point to understand is that VI priorities are in no way associated with thread priorities. The thread that pulls it off the run queue will execute a VI with high priority. If the thread has background priority, a slow thread will execute the high-importance VI. The execution engine will not take a task away from one thread and reassign it to a thread with a more suitable thread priority.

To help illustrate the use of run queues, consider the in box on your desk. With LabVIEW 4.1 and earlier, there was a single in box, and each time a VI was able to run it would be put into the in box and you would be able to grab the task and perform it. LabVIEW 5.0 and later have multiple run queues which equate to one in box for each of your hands. As tasks become available they get put into an appropriate in box and the hand that corresponds to that in box can grab the task. Another comparison for Windows programmers is the message pump. Windows 3.1 had a single message loop. Each time an event occurred, such as a mouse click, the

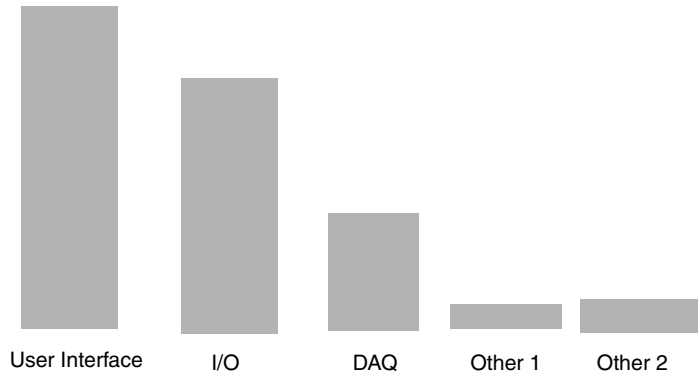


FIGURE 9.7

event would be put into the system-wide message loop. All applications shared the same message loop, and numerous problems were caused because some applications would not return from the message loop and would lock up windows. Windows 95 and later have message loops for each application. Every application can continue to run regardless of what other applications are doing. We have the same benefit in LabVIEW. VIs assigned to different subsystems can now operate with their own threads and their own run queues.

A thread will go to the run queue associated with its subsystem and pull the top task off the list. It will then execute this task. Other threads in the subsystem will go to the run queue and take tasks. Again, this is not a FIFO stack — the highest-priority VI will be handed to a thread. This leaves a lot of room for both performance tweaking and performance degradation. Priorities other than "normal" should be the exception, and not status quo.

When a VI is configured to run only in a particular subsystem, it will be put onto the run queue of that particular subsystem, and then the VI must wait for a thread belonging to this system to be assigned to execute it. This can cause performance degradation when thread priorities are different between subsystems. Section 9.8.2 discusses thread configurations in multisubsystem LabVIEW applications.

Figure 9.7 shows the run queues that are developed when a LabVIEW code diagram is run. When VIs are scheduled to run in the "same as caller subsystem," a thread belonging to the subsystem will end up executing the VI. A subtle point is that if there are multiple threads belonging to the subsystem, there are no guarantees which thread will execute which VI.

9.8.3 DLLs in Multithreaded LabVIEW

Special care must be taken when working with DLLs in multithreaded LabVIEW. DLLs can potentially be called from several different threads in LabVIEW. If the DLL has not been written to handle access by multiple threads, it will likely cause problems during execution. Recall thread safe in Section 9.1.10. If mutexes, semaphores, or critical sections are not explicitly designed into the DLL, then it is not guaranteed to be thread safe.

Threading problems are not always obvious. Several million calls may need to be made to the DLL before a problem surfaces. This makes troubleshooting thread problems extremely difficult. Bizarre program operation can be extremely difficult to troubleshoot, especially when the code can execute for days at a time without failure. When working with DLLs, and crashes occur only occasionally, suspect a thread problem.

When writing C/C++ code to be called by LabVIEW, you need to know if it will possibly be called by multiple threads of execution. If so, then you need to include appropriate protection for the code. It is fairly simple to provide complete coverage for small functions. The Include file that is needed in Visual C++ is process.h. This file contains the definitions for Critical Sections, Mutexes, and Semaphores. This example is fairly simple and will use Critical Sections for data protection. A Critical Section is used to prevent multiple threads from running a defined block of code. Internal data items are protected because their access is within these defined blocks of code. Critical Sections are the easiest thread protection mechanism available to the Windows programmer, and their use should be considered first.

```
#include <process.h>

//Sample code fragment for CriticalSections to be
used by a //LabVIEW function.

CRITICAL_SECTION Protect_Foo

Void Initialize_Protection(void)

{

  INITIALIZE_CRITICAL_SECTION(&Protect_Foo);

}

Void Destroy_Protection(void)

{

  DELETE_CRITICAL_SECTION(&Protect_Foo);

}

int foo (int test)

{

  int special_Value;

  ENTER_CRITICAL_SECTION(&Protect_Foo); //Block
      other threads from accessing

  Special_Value = Use_Values_That_Need_
      Protection(void);

  LEAVE_CRITICAL_SECTION(&Protect_Foo);//Let other
      threads access Special Value, I'm finished.
```

```
    Return special_Value;
}
```

The fragment above does not do a lot of useful work as far as most programmers are concerned, but it does illustrate how easy thread protection can be added. When working with Critical Sections, they must be initialized prior to use. The INITIALIZE_CRITICAL_SECTION must be called. The argument to this function is a reference to the Critical Section being initialized. Compile errors will result if the Critical Section itself is passed. The Critical Section must also be destroyed when it will no longer be used. Initialization and destruction should be done at the beginning and end of the application, not during normal execution.

Using a Critical Section requires that you call the Enter and Leave functions. The functions are going to make the assumption that the Critical Section that is being passed was previously initialized. It is important to know that once a thread has entered a Critical Section, no other threads can access this block until the first thread calls the Leave function.

If the functions being protected include time-consuming tasks, then perhaps the location of the Critical Section boundaries should be moved to areas that access things like data members. Local variables do not require thread protection. Local variables exist on the call stack of the thread and each thread has a private call stack, which makes local variables completely invisible to other threads.

C++ programmers must also remember that LabVIEW uses C naming conventions. The keyword extern C must be used when object methods are being exposed to LabVIEW. LabVIEW does not guarantee that C++ DLLs will work with LabVIEW. In the event you encounter severe problems getting C++ DLLs to operate with LabVIEW, you may have hit a problem that cannot be resolved.

A few additional notes on DLLs being called from LabVIEW before we complete this section.

LabVIEW uses color-coding to identify DLLs that are executed by the user interface thread from DLLs that are listed as "re-entrant." If a DLL call is shown in an orange icon, this identifies a DLL call that will be made from the User Interface subsystem. If the standard off-yellow color is shown, it will be considered re-entrant by LabVIEW and will allow multiple threads to call the DLL. Library call functions default to User Interface subsystem calls. If a DLL was written to be thread safe, then changing this option to reentrant will help improve performance. When a User Interface Only DLL call is made, execution of the DLL will wait until the user interface thread is available to execute the call. If the DLL has time-consuming operations to perform, the user interface's performance will degrade.

When working with DLLs of questionable thread safety, always call them from the user interface. When it is known that threading protection has been built into a DLL, make the library call re-entrant. This will allow multiple threads to call the DLL and not cause performance limitations. If you are stuck with a DLL that is not known to be thread safe, be careful when calling the DLL from a loop. The number of thread context switches will be increased, and performance degradation may set in. We did get a tip from Steve Rogers, one of the LabVIEW development team

members, on how to minimize the number of context switches for DLLs. This tip works when you are repeatedly calling a DLL from a loop that is not assigned to the user subsystem. Wrap the DLL call in a VI that is assigned to the user subsystem. The thread context switches have been moved to the VI call and not the DLL call. Effectively, this means that the entire loop will execute in the user subsystem, and the number of needed context switches will drop dramatically.

Execution of a DLL still blocks LabVIEW's multitasking. The thread that begins executing the DLL will not perform other operations until the DLL call has completed. Unlike LabVIEW 4.0 and earlier, other threads in the subsystem will continue to perform tasks.

9.8.4 Customizing the Thread Configuration

The number of threads in each LabVIEW execution subsystem is specified in the labview.ini file. This information should not be routinely altered for development workstations. The default configuration will work just fine for most applications. The default configuration for a multithreaded platform is four threads of execution per subsystem of normal priority.

In situations where a LabVIEW application is running on a dedicated machine, tweaking the thread configuration can be considered. National Instruments provides a useful VI for configuring the ini file used by LabVIEW: threadconf.vi. The Thread Configuration VI should be used to alter LabVIEW's thread configuration. Vendors are expected to supply tools to easily modify application configurations. The most difficult aspect of using this VI is locating it! The VI can be found in vi.lib\utilities\sysinfo.llb. Figure 9.8 shows the front panel of this VI.

To change the configuration of the thread engine, select Configure. A second dialog box will appear, as shown in Figure 9.8. Each execution subsystem may be configured for up to eight threads per priority. Some readers may feel compelled to review Section 9.5. A plethora of options are available, and the advice is not to alter most of them. In the next section, information on estimating the maximum number of useful threads is presented.

A few words about thread priorities: Time-critical priorities are, in general, hazardous. Consider Windows XP and time-critical threads. Mouse clicks and keypad activity are reported to the operating system; the operating system then sends a message to the application that should receive the user input. In the event that time-critical threads are running, they may take priority over operating system threads. The commands you enter to Quit may never arrive, and the application will have to be killed (end task from task manager).

Background priorities may be useful but, in general, keeping all threads at normal priority is best. If all threads are running on normal priority, none of them will suffer from starvation or priority inversion. The easiest way to avoid complex threading issues is to avoid creating them. This is the biggest caveat in this section; review Section 9.5 for descriptions on priority inversion and starvation. If more threads are available than can be used, LabVIEW's execution engine will allow the thread to continue checking the queue for VIs that belong to its subsystem. This will require a minimal amount of CPU time and avoids thread problems.

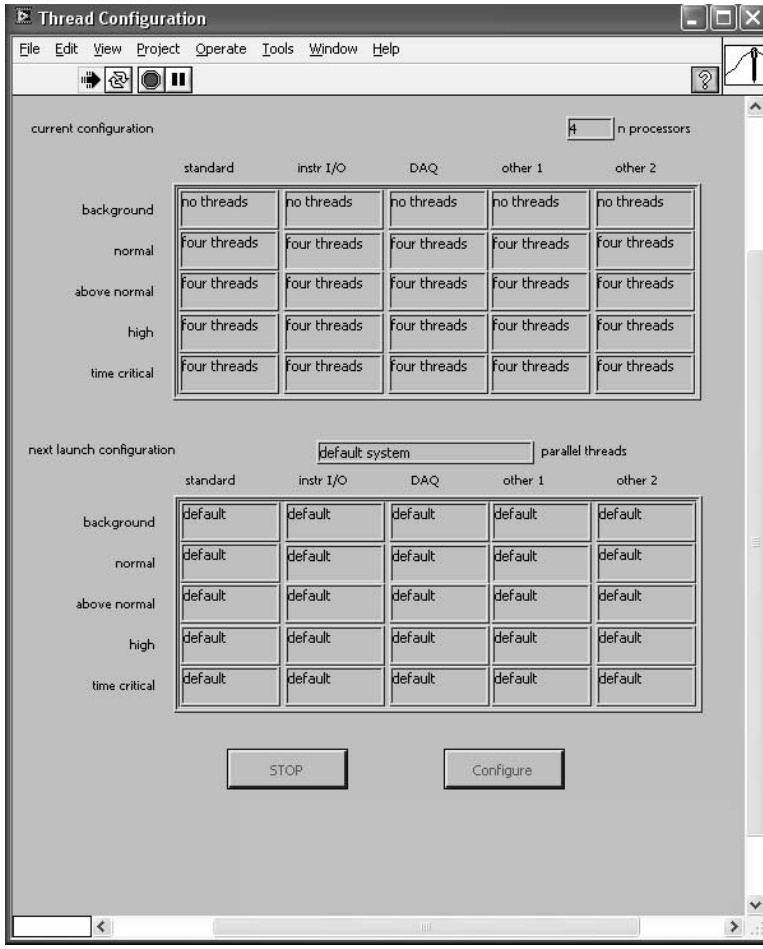


FIGURE 9.8

In a Hyper-Threading environment, the CPU will shut down the second CPU core if only one thread is actively running. Having excessive threads will potentially hinder the CPU from optimizing its performance. LabVIEW will start only the threads specified for the user subsystem when it starts up. The reason is to minimize system resource usage. We do not want to create more threads than necessary because that would cause extra overhead for the operating system to track. LabVIEW defers creation for other subsystems' threads when it loads a VI that is assigned to a subsystem into memory. Performance tuned applications may want to run an empty VI for each subsystem in use during application startup. This allows for the application to create the needed threads in advance of their usage. The LabVIEW INI file specifies the number of threads created for any particular subsystem.

Windows users may see an additional thread created when dialog boxes are used. The dialog boxes for Printers, Save File, Save File As, and Color Dialog are actually

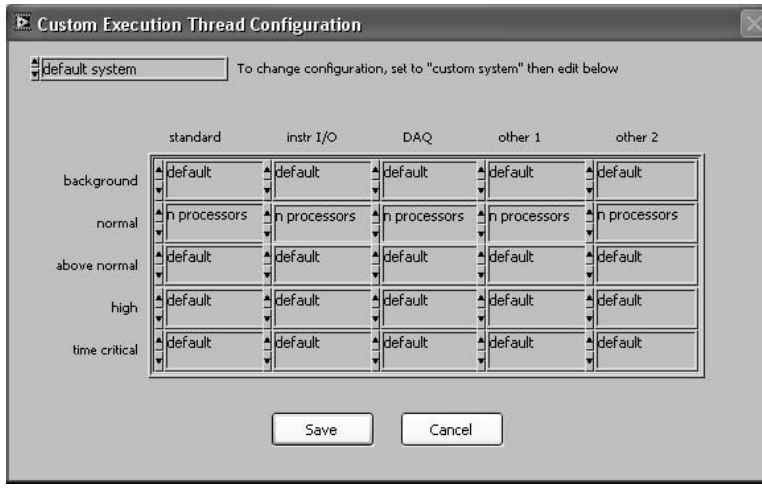


FIGURE 9.9

the same dialog box. This dialog box is called the common dialog and is responsible for creating that extra thread. This is normal operation for Windows. The common dialog thread is created by the dialog box, and is also destroyed by the dialog box. High-performance applications should refrain from displaying dialog boxes during execution except when absolutely necessary.

When working with the Application Builder, the thread information will be stored in a file that is located with the executable generated by LabVIEW. The same rules apply to application builder programs as mentioned above. Normal priority threads are all that a programmer should ever need; do not create more threads than an application can use. Recall that the existence of threads translates to performance gains only when multiple processors are available on the machine.

Another word of caution when configuring the threads LabVIEW uses: since LabVIEW is an application like any other running on the system, its threads are scheduled like any other, including many operating system components. When LabVIEW threads are all high priority, they just might steal too much time from the operating system. This could cause system-wide inefficiency for the operating system, and the performance of the entire computer will become degraded or possibly unstable. The fundamental lesson when working with thread priorities is that your application is not the only one running on the system, even if it is the only application you started.

9.9 THREAD COUNT ESTIMATION FOR LABVIEW

When discussing thread count, this section will refer exclusively to the maximum number of useful threads, which is the number of threads that will be of benefit to the execution systems without any threads being left idle. The minimum number of useful threads to the LabVIEW execution engine is always one.

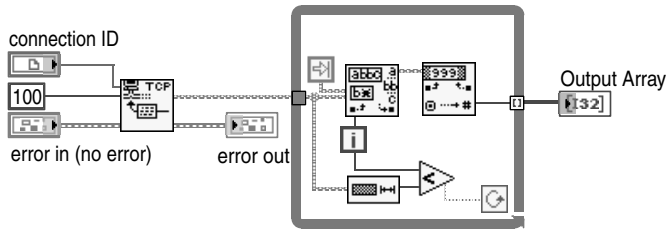


FIGURE 9.10

Consider the VI code diagram presented in Figure 9.10. The number of useful threads for this VI is one. This VI obviously does not consider events such as errors, fewer than 100 bytes read, or the spreadsheet String-to-Array function. In spite of the fact that it is not well-thought-out, it is an excellent dataflow example. Everything will happen in the VI in a well-established order: the TCP Read function will execute, and then the While loop will execute. There is only one path of execution, and multiple threads will do nothing to optimize the execution of this VI. The fact that the While loop will execute multiple times does not suggest multiple threads can help. It does not matter which thread in which subsystem is currently processing this VI; execution will still happen in a defined order. Thread context switches will do nothing to help here. The lesson learned in this simple example: if order of execution is maintained, multithreading is not going to improve application performance.

Alert readers would have noticed the location of the String Length function. In the While loop it is possible for two threads to perform work, but one will only need to return a string length, which is a very simple function. This is not a significant amount of work for a thread to do. Also, it would be far more efficient to locate the String Length function outside the While loop and feed the result into the loop. When optimizing code for threading, look for all performance enhancements, not just the ones that impact threading potential. Having an additional thread for this example will not improve performance as much as moving the string length outside the While loop.

The VI code diagram presented in Figure 9.11 presents a different story. The multiple loops shown provide different paths of execution to follow. Threads can execute each loop and help maintain the illusion that all loops seem to execute at the same time. If multiple CPUs are involved, application speed will improve. The number of useful threads for this example is three. If the internal operations of the loops are time-intensive operations, then a fourth thread may be desirable. This fourth thread will be added to help support the front panel. If there are graphs or other intensive operations, consider an additional thread for supporting the display. Recall that additional threads will take some time away from other threads.

Now consider the VI code diagram shown in Figure 9.12. There appears to be a single path for this VI to take, but considering that several of the VIs are not waiting on inputs, they can be scheduled to run right away. There are four paths of execution that merge into one at the end of the code diagram. Threads can help here, and the maximum number of useful threads will be equal to the number of paths of execution. In this case, the maximum number is four. If several of these subVIs are

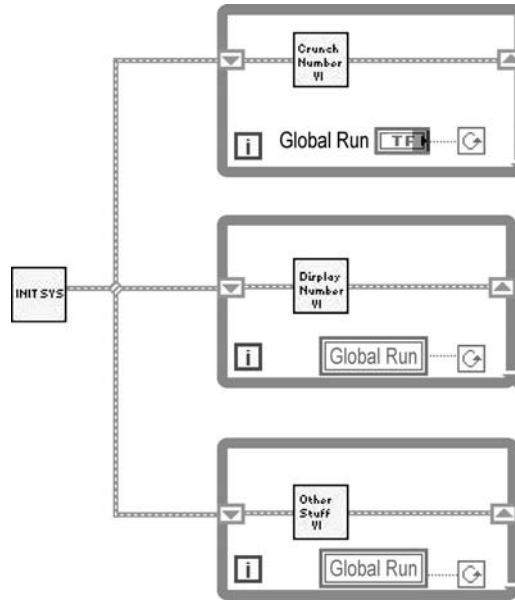


FIGURE 9.11

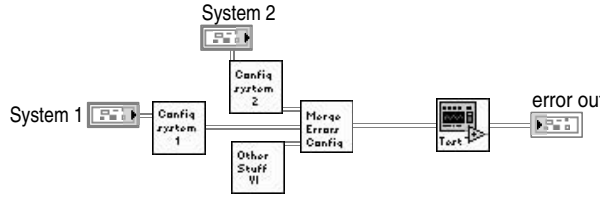


FIGURE 9.12

expected to execute quickly, then they do not require a thread to exist on their behalf, and you should consider reducing the number of threads. Recall that each thread is going to be scheduled and requires some time on the CPU. The lower the thread count, the more execution time per thread.

The code diagram presented in Figure 9.13 is basically a mess. Multiple threads could potentially be beneficial here, but if the operations splattered about the display were modularly grouped into subVIs, then the benefit seen in Figure 9.12 would still exist. You can consider prioritizing the subVIs as subroutines; the benefit of reduced overhead would make the VI run nearly as fast, and a lot of readability will be gained. Section 9.9 describes criteria for using subroutine VIs in multithreaded LabVIEW.

We have gone through a few simple examples concerning only a single VI. When a large-scale application is going in development, the maximum number of useful threads will probably not skyrocket, but the determination can be much more difficult. An application consisting of 250 subVIs will be time-intensive for this

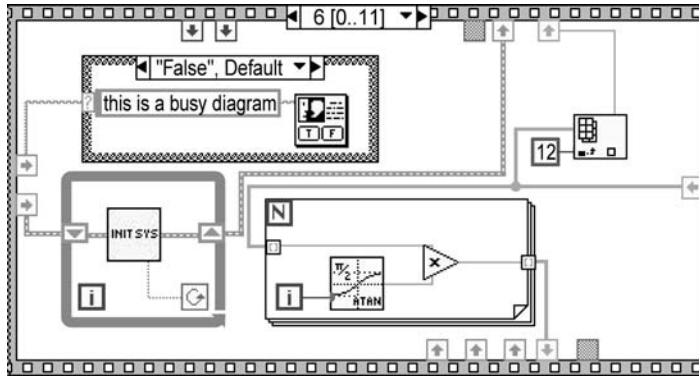


FIGURE 9.13

type of analysis. The programmer's intuition will come into play for application-wide analysis. Also, never forget that at some point, adding threads is not going to make an improvement. Unless you are working with a Hyper-Threading quad-CPU system, having hundreds of threads of execution is not going to buy much in terms of performance.

9.9.1 Same as Caller or Single Subsystem Applications

When attempting to determine the maximum number of useful threads for an application, the maximum number of execution paths for the application must be determined. This can be difficult to accomplish. For example, look at the hierarchy window of a medium to large-scale application. Each branch of a hierarchy window does not equate to one branch of execution. A programmer who is familiar with the functionality of each subVI will have an understanding of the tasks performed in the subVI. Look at subVIs that have descriptions that suggest parallel operations. This is a difficult piece of advice to generalize, but programmers should be familiar with their application. Order of execution may be forced in a number of branches, and this will limit the number of useful threads. Having more threads than necessary will cause minor performance hits. The number of thread context switches that will be incurred when threads are given time will be increased. If the thread configuration includes threads of differing priority, then lower-priority threads may receive little execution time and not be of much help to the application.

The simplest case to analyze is when all threads are running in a single LabVIEW subsystem or all VIs are assigned to a "same as caller" subsystem. Then there are no needed considerations to be made regarding which subsystems require threads. On a system dedicated to running an application of this type, consider modifying the thread configuration so that only one subsystem has threads — the User subsystem.

The following VI code diagram simply demonstrates a main level VI and its three independent loops. Obviously, three threads may support this VI. If the three loops require heavy numerical processing, then a fourth thread may be desired if a lot of display updates are also desired. Since the three subVIs are going to keep busy running numerical calculations, a fourth thread could be brought in for GUI

updates. Understand that LabVIEW is not going to allocate a thread to each loop and the fourth to the VI, but there will always be four threads looking into the run queue for a new task. If threads take a lot of time to complete one iteration of a loop, then three threads may periodically become bogged down in a loop. The fourth thread exists to help out when circumstances like this arise. When no intensive GUI updates are required, the fourth thread is not desirable. Additional thread context switches can be avoided to improve performance.

Reconsidering the above example, if one of the loops performs a very simple operation, then reducing the number of threads to two may also be beneficial. Having fewer threads means less work for the operating system to do. This is a judgment call the programmer is going to have to consider. The fundamental trade-off is going to be parallel operation versus operating system overhead. The general guideline is to have fewer threads and minimize overhead. When looking to estimate the number of threads, look for operations that are time consuming. Examples of time-consuming operations are large array manipulation, DLL calls, and slow data communications, such as serial ports.

9.9.2 Multiple Subsystem Applications

Determining how many threads can support a LabVIEW application with VIs running in dedicated subsystems requires additional work. The number of useful threads per subsystem must now be considered. The solution to this problem is to analyze the number of paths of execution per subsystem. Considerations must be made that threads may become blocked while waiting for VIs to be assigned to other subsystems. Again, an additional thread may be considered to help out with display updates. Do not forget that an additional thread will take some time away from other threads in LabVIEW. High-performance applications may still need to refrain from displaying graphs during run-time.

It is still possible to write many multithreading optimized applications without resorting to using multiple subsystems. LabVIEW's configuration allows for a maximum of eight threads per subsystem. When you conclude that the maximum number of useful threads is well beyond eight, then forcing some VIs to execute in different subsystems should be considered. If fewer than nine threads can handle a VI, do not force multiple subsystem execution. Performance limitations could arise with the extra contact switching.

A special case of the multiple subsystem application is a distributed LabVIEW application. Optimization of this application should be handled in two distinct parts. As LabVIEW is executing independently on two different machines, you have two independent applications to optimize. Each machine running LabVIEW has two separate processes and each will have their own versions of subsystems. When the threading model is being customized, each machine should have its own threading configuration. One machine may be used solely for a user interface, and the other machine may be executing test or control code. Consider using the standard configuration for the user interface machine. It is unlikely that sophisticated analysis of the user interface is required. Consider investing engineering time in the more important task of the control code. In situations where each instance of LabVIEW

is performing some hard-core, mission-critical control code, both instances of LabVIEW may have their threading configurations customized.

Your group should deploy a coding standard to indicate information regarding the subsystem a VI is assigned to. When trying to identify problems in an application, the subsystem a VI is assigned to is not obvious. A programmer must actively look for information regarding that. A note above or below the VI should clearly indicate that the VI has been forced into a subsystem. An alternative to using notes is to color-code the icon, or a portion of it, to clearly indicate that the VI has been forced into a nonstandard mode of execution. This will simplify debugging and maintenance. Multiple subsystem applications will almost always be very large-scale applications; these types of techniques will simplify maintenance of such large applications.

9.9.3 Optimizing VIs for Threading

When you are writing code for which you would like to have the maximum benefit of the threading engine, avoid forcing the order of execution whenever possible. When a VI is coded for tasks to happen in a single-file fashion, the tasks assigned to the run queue must also be assigned in a single-file fashion. This limits the ability of the threads to handle tasks because they will always be waiting for a task to become available. If possible, avoid the use of sequences; they are going to force an order of execution. Let the error clusters force an order of execution for things like read and write operations. Operations that are handled, such as loading strings into a VI and determining the value of some inputs, can be done in a very parallel fashion. This will maximize the ability of the threads to handle their jobs. All simple operations will have their data available and will be scheduled to run.

As an example of maximizing data flow, consider the code diagram in Figures 9.14, 9.15, and 9.16. These three diagrams describe three sequences for a simple

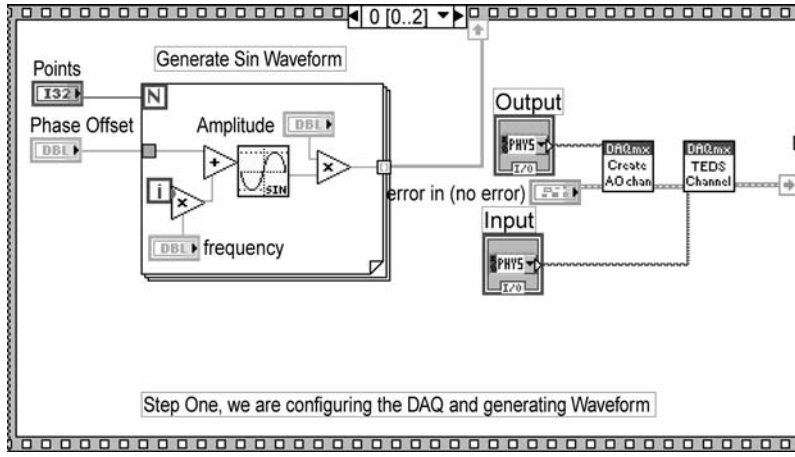


FIGURE 9.14

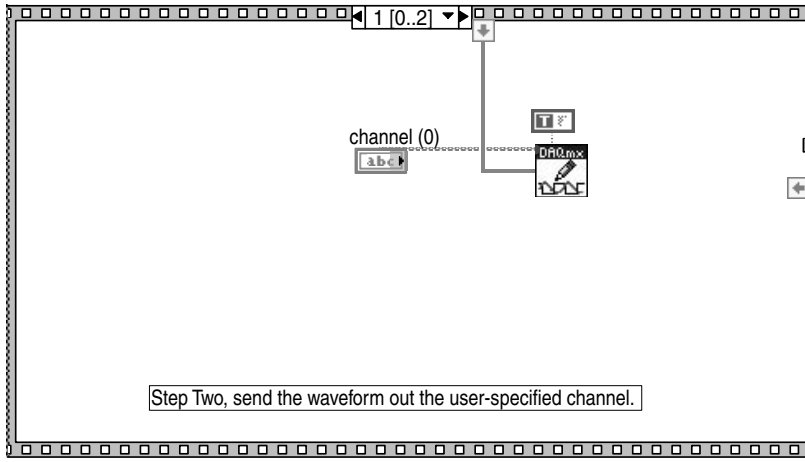


FIGURE 9.15

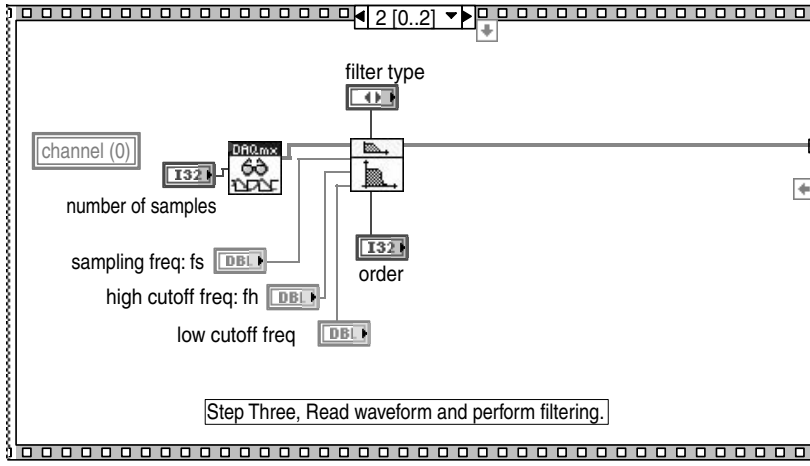


FIGURE 9.16

data acquisition program. The items in the first sequence must be handled and completed before the second can be executed. The second sequence is fairly simple, and the waveform is shipped out. The third sequence reads in a signal and filters it. The DAQ experts may criticize the appearance of this VI, but it serves as an example of how sequences limit the thread's ability to operate.

In the first sequence there are two paths of execution to follow. The first is the generation of the sine waveform to be used. The second path to follow is the Analog Output and Analog Input VIs. Please note that the error cluster forces an order of execution; the Output VI must be executed, then the Input VI. There is some initial loading of values on the wire table that needs to be done. The threads will also handle this.

The second sequence diagram simply sends out the waveform. The inputs here cannot be processed and moved on the wire table until this sequence starts executing. Had this VI been in the first sequence, the constants could have already been shifted in LabVIEW's wire table.

The third sequence reads an input waveform and runs it through a Butterworth filter. Many DAQ experts will argue about the timing delays and choice of a Butterworth filter, but we are putting emphasis on the threading issues. The constants in this sequence also may not be loaded into new sections of the wire diagram until this sequence begins execution.

Let us quickly rethink our position on the number of paths that could be followed in the first sequence. Two was the decided number, one for the signal generation, and one for the Configuration VIs. Recall the Setup VIs have multiple subVIs with the possibility of dozens of internal paths. We are unable to maximize the number of executable paths because the order of execution is strongly forced. The "thread friendly" version is shown in Figure 9.17. Wrapping the Output Generation VI in a sequence was all that was needed to force the Configuration, Generation, and Read functions. The one-step sequence cannot execute until the error cluster output becomes available.

The Configuration VIs are set in parallel with a little VI inserted to add any errors seen in the clusters. This is a handy little VI that is included on the companion CD to this book. The multiple execution paths internal to these VIs are now available to the threading engine.

All constants on the block diagram can be loaded into appropriate slots on the wire table without waiting for any sequences to start. Any of these functions can be encapsulated into subVIs to make readability easier. VIs that easily fit on a 17-in. monitor should not require 46-in. flat-panel displays for viewing after modification.

The lesson of this example is fairly simple: do not force order of execution in multithreaded LabVIEW. If you want to take full advantages of the threading engine, you need to leave the engine a little room to have execution paths. Obviously, some order must exist in a VI, but leave as many execution paths as possible.

This next part of optimization has less to do with threads than the above example, but will stress good programming practice. Polling loops should be minimized or eliminated whenever possible. Polling loops involve some kind of While loop continuously checking for an event to happen. Every time this loop is executed, CPU cycles are burned while looking for an event. In LabVIEW 4.1 and earlier versions, you may have noticed that the CPU usage of your machine ran up to 100%. That is because the polling loop was "tight." Tight loops do very little in a cycle. This allows the loop to complete its execution quickly and take more time. Because there is always something in LabVIEW's run queue to do (run the loop again), LabVIEW appears to be a very busy application to the system. LabVIEW will get all kinds of time from the scheduling algorithms, and the performance of the rest of the system may suffer. In LabVIEW 5.0 the threads that are assigned tasks for the loop will be just as busy, and therefore make LabVIEW again look like a very busy application. Once more, LabVIEW is going to get all kinds of time from the operating system which will degrade performance of the system.

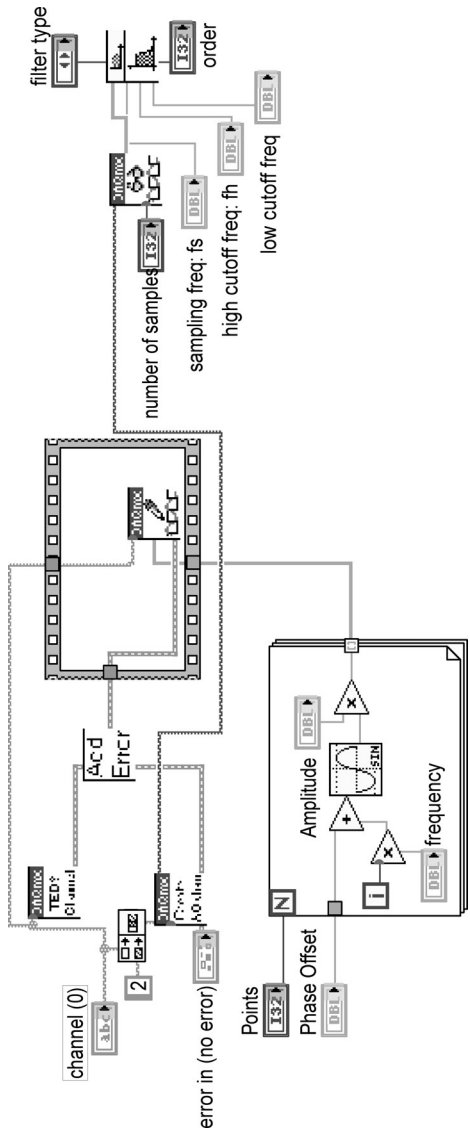

filter type
order
low cutoff freq
high cutoff freq: fh
sampling freq: fs
number of samples
Acd
Error
channel (0)
error in (no error)
Points
Phase Offset
Amplitude
frequency
SIN
2

FIGURE 9.17

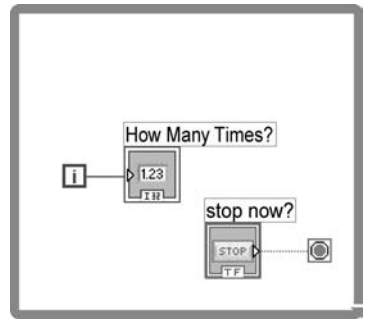


FIGURE 9.18

Figure 9.18 shows a simple loop that increments a counter. This is a short example of a tight loop. There is very little activity going on inside the loop, and millions of iterations happen in very little time. If the value of the loop iterator is wired to a terminal, the execution speed will slow down because of the volume of graphics updates that need to happen. The System Monitor (Windows 95), Task Manager (Windows XP), or an application such as Top or Monitor (UNIX) will show a significant amount of CPU usage. The problem is there isn't much useful happening, but the amount of CPU usage will top out the processor. Other applications will still get time to run on the CPU, but they will not receive as much time because the tight loop will appear to always need time to run when viewed by the operating system scheduler.

Tight loops and polling loops cannot always be avoided. When an application is looking for an external event, polling may be the only way to go. When this is the case, use a Wait Milliseconds command if possible. It is unlikely that every polling loop needs to check a value at CPU speeds. If this is the case, the selection of LabVIEW may be appropriate only on Concurrent PowerMAX systems, which are real-time operating systems. If the event does not absolutely need to be detected, make the loop sleep for a millisecond. This will drastically reduce the CPU utilization. The thread executing the wait will effectively be useless to LabVIEW while sleeping, but LabVIEW's other threads do not need to fight for CPU time while the thread is executing the tight loop.

When waiting on events that will be generated from within LabVIEW, using polling loops is an inefficient practice. Occurrence programming should be used for this. This will prevent any of LabVIEW's threads from sitting in polling loops. The code that is waiting on an occurrence will not execute until the occurrence is triggered. No CPU cycles are used on nonexecuting code.

9.9.4 Using VI Priorities

The priority at which the VI executes may also be considered when configuring VIs. The VI execution priority is not directly related to the threads that execute the VI, or the priority of the threads that execute the VI. Figure 9.19 shows the configuration of a VI. The priority levels assigned to a VI are used when LabVIEW schedules

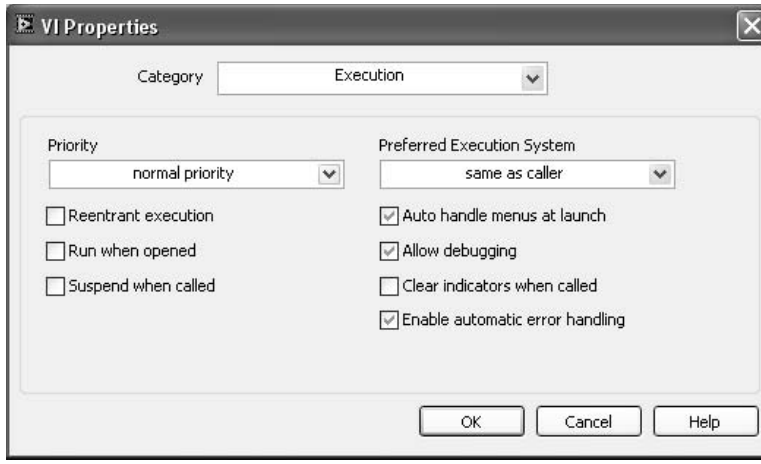


FIGURE 9.19

tasks in the run queue. The priority of the VI has nothing to do with the priority of the thread that is executing it. When a thread with high priority is assigned a VI with background priority, the thread will not reduce its priority to accommodate the VI. The background importance VI will be executed with blazing speed. The reverse is also true.

When working with VI priorities, recall multithreading problem definitions; several of them can be caused in LabVIEW's scheduling routines. Starvation is the easiest problem to cause. When a VI is listed as background priority, VIs with higher priorities will be put into the run queue ahead of the low-priority VI. This will cause the execution of the low-priority VI to be delayed. This could impact the performance of the code diagram. What will end up happening is that all other eligible tasks will be run until the low-priority VI is the only available task to be executed. This would form a bottleneck in the code diagram, potentially degrading performance. The use of VI priorities should not be used to force the order of execution. Techniques using error clusters should be used instead. LabVIEW's engine makes no promises regarding execution time, much like a multithreaded operating system's scheduling algorithm. In the event that parallel executing loops are involved, it is possible for the background priority VI never to be executed.

Priority inversion can also be caused by VI priorities. Recall that the priority of a VI does not impact or change the priority of the thread(s) executing it. If a VI with high priority depends on the outputs of a VI with lower priority, execution of the high-priority VI will be delayed until the low-priority VI has completed execution. This potential for performance limitations should be avoided.

Race conditions can also be induced with VI priorities. The threading model used does not induce these race conditions. These would be race conditions caused by the code diagram itself.

The best logic to use to prevent VI priority problems is similar to preventing problems with the threading engine. A priority other than "normal" should be an

exception, not the norm. If a convincing case cannot be put into the Description of Logic of the VI, then its execution priority should be normal. In general, we avoid changing VI priorities. Forcing the order of execution is a better mechanism to accomplish control of a code diagram. In addition, it is much easier for a programmer to look at a VI and understand that the order of execution is forced. VI priority is somewhat hidden in the VI's configuration; a programmer must actively search for this information. Assuming that programmers will examine your code and search the configuration is unwise; most people would not suspect problems with VI priority.

As a coding standard, when a VI has an altered priority, a note should be located above or below to clearly indicate to others who may use the VI that there is something different about it. Another flag that may be used is to color-code the icon or portion of the icon indicating that its priority is something other than normal.

If you absolutely insist on keeping a VI as a priority other than normal, then use the following tip from Steve Rogers (LabVIEW developer extraordinaire): VIs of high priority should never be executing continuously. High-priority VIs should be kept in a suspended mode, waiting on something such as an occurrence, before they are allowed to execute. Once the VI completes executing, it should be suspended again and wait for the next occurrence. This allows for the high-priority VI to execute as the most important VI when it has valid data to process, and to not execute at all when it is not needed. This will prevent programmers from creating priority inversion or starvation issues with LabVIEW's run queue management.

9.10 SUBROUTINES IN LABVIEW

As hinted throughout the chapter, subroutine VIs have strong advantages when using multithreading. First, we need to review the rules on subroutine priority VIs:

1. Subroutine VIs may not have a user interface.
2. Subroutine VIs may call only other subroutine-priority VIs.
3. Subroutines may not call asynchronous nodes (dialog boxes, for example; nodes that do not have a guaranteed return time).

It is important to understand that subroutine classification is not a true priority. "Subroutine" denotes that this VI is no longer a standard VI and that its execution and compilation are radically different from other VIs. Subroutine priority VIs do not have a priority associated with them, and they are never placed into the run queues of LabVIEW. Once all inputs for the subroutine are available, the subroutine will execute immediately, bypassing all run queues. The subsystem associated with the subroutine will stop processing tasks until the subroutine has completed execution. This might sound like a bad idea, but it is not. Having a routine complete execution ASAP is going to get its operation over as quickly as possible and allow LabVIEW to do other things fairly quickly. Subroutines are a bad idea when very time-intensive tasks need to be done because you will block the run queue for a subsystem for an extended amount of time.

Subroutines execute faster than standard VIs because they use less overhead to represent instructions. You may not have a user interface on subroutine priority VIs

because, technically, a subroutine does not have a user interface. This is part of the reduced overhead that subroutines have.

Subroutine VIs may call only other subroutines because they are executed in an atomic fashion. Once execution of a subroutine VI starts, single-threaded LabVIEW execution engines will not do anything else until this subroutine has finished. Multitasking becomes blocked in single-threaded LabVIEW environments. Multithreaded LabVIEW environments will continue multitasking when one thread enters a subroutine. The thread assigned to work on the subroutine may do nothing else until the subroutine is executed. Other threads in the system are free to pull jobs off the run queue. In the next section, we will discuss the data types that LabVIEW supports; this is relevant material when subroutine VIs are considered.

9.10.1 Express VIs

With LabVIEW 7's release a new type of programming construct became available, namely express VIs. These VIs contain precompiled code that accepts inputs. The question with respect to this chapter is how do express VIs interact with LabVIEW's thread model.

As far as LabVIEW's execution engine is concerned, an express VI is an atomic function, meaning it is equivalent to a subroutine. Once an express VI has been put into a run queue, the thread that begins to execute it will finish execution before it can process other tasks.

In the event an express VI is broken out into editable code, then it is handled as any other VI with respect to thread count estimation.

9.10.2 LabVIEW Data Types

Every LabVIEW programmer is familiar with the basic data types LabVIEW supports. This section introduces the low-level details on variables and data storage. Table 9.1 shows the LabVIEW data types. Of concern for application performance is how fast LabVIEW can process the various data types. Most numerical processing can always be assumed to be relatively fast.

As stated in Table 9.1, Booleans are simply 16-bit integers in LabVIEW 4.0 and earlier, and 8-bit integers in LabVIEW 5.0 and later. Their storage and creation is fairly quick; arrays of Booleans can be used with minimal memory requirements. It must be noted that computers are minimum 32-bit machines, and are transitioning to 64 bit access. Four bytes is the minimum amount of memory that can be addressed at a time. One- and two-byte storage is still addressed as four-byte blocks, and the upper blocks are ignored.

Integer sizes obviously depend on byte, word, or long word selections. Integer arithmetic is the fastest numerical processing possible in modern hardware. We will show in the next section that it is advantageous to perform integer processing in one thread of execution.

Floating-point numbers also support three precision formats. Single- and double-precision numbers are represented with 32- or 64-bit numbers internal to LabVIEW. Extended precision floating-point numbers have sizes dependent on the platform you are using. Execution speed will vary with the types of operations

TABLE 9.1
LabVIEW Data Types

Data Type	Size	Processing Speed	Notes
Boolean	16 bits (LabVIEW 4) 8 bits (LabVIEW 5), high bit determines true/false	Fast	High bit determines true or false.
Integers	8, 16, 32, 64 bits	Fast	Signed and Unsigned
Floating Point	Depends on type and platform	Fast	Extended precision size is machine-dependent; single and double are 32- and 64-bit numbers
Complex	Depends on type and platform	Medium?	Slower than floating points.
String	4 bytes + length of string	Slow	First 4 bytes identify length of string
Array	Variable on type	Slow	Faster than strings but can be slow, especially when the array is dimensioned often.
Cluster	Depends on contents	Slow	Processing speed depends heavily on contents of cluster.

performed. Extended-precision numbers are slower than double-precision, which are slower than single-precision numbers. In very high performance computing, such as Software Defined Radio (SDR) where large volumes of floating point data need to be processed in fairly short order, a tradeoff of numerical precision versus processing time is made quite often. The lower precision numbers contribute to quantization noise, but in some cases make or break the performance of an application. Floating-point calculations are always slower than integer arithmetic. Each floating point stores sections of a number in various parts of the memory allocated. For example, one bit is used to store the sign of the number, several bytes will be used to store the mantissa, one byte will store the sign of the exponent, and the rest will store the integer exponent of the number. The format for single- and double-precision numbers is determined by National Instruments, and they are represented internally in LabVIEW. Extended-precision number formats depend on the hardware supporting your system.

Complex numbers use a pair of floating-point numbers for representation. Complex numbers use the same precision as floating-point numbers, but they are slower for processing. Each complex multiplication involves four floating-point calculations. Additions and subtractions involve two floating-point calculations. When necessary, complex calculations need to be done, but their execution speed must be considered in performance-critical applications.

String processing can be very slow. LabVIEW uses four bytes to indicate the length of the string internally, and the contents of the string following the length preamble. This is an advantage LabVIEW programmers have over their C counterparts. C style strings must end with an ASCII 0 (NULL); these NULL-terminated strings assume that there are no NULL values occurring in the middle of the string.

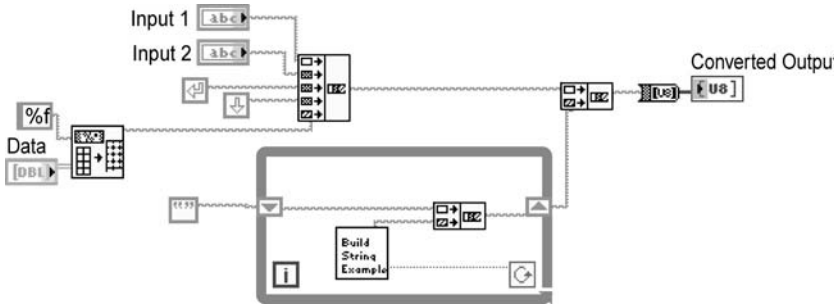


FIGURE 9.20

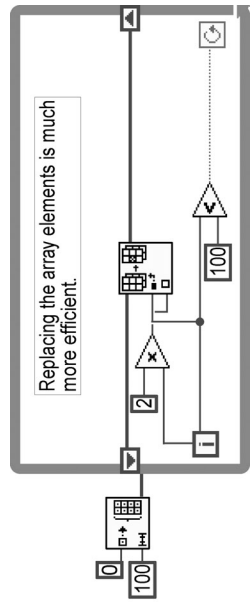
Microsoft compilers also use a string construct called the "BSTR" which looks a lot like LabVIEW's string, 4 bytes of length information preceding the character data. Oddly, most of the BSTR support functions still are not capable of supporting embedded NULLs. LabVIEW strings do not have this no-embedded NULLs requirement. This is advantageous when working with many devices and communications protocols.

Any time you perform an operation on a string, a duplication of the string will be performed. In terms of C programming, this will involve a "memcopy." BSTR types or OO string classes still use a memory copy; the memcopy is abstracted from the programmer. Memory copies involve requesting an allocation of memory from the memory manager and then duplicating the memory used. This is a performance hit, and, although it cannot be entirely avoided, performance hits can be minimized. Whenever possible, major string manipulation should be avoided when application performance is required. Examine Figure 9.20 for an illustration for where memory copies are made. Memory copies will be made for other variable types, but sizes for integers are 4 bytes, floating points are a maximum of 8 bytes, and Booleans require a minimum of 32 bits for storage. The shortest string representation in LabVIEW is an empty string, which requires five bytes, the four-byte preamble, and one blank byte. Most strings contain information, and longer strings require more time to copy internally.

Array processing can be significantly faster than string processing, but can also be hazardous to application performance. When using arrays in performance-critical applications, pre-dimension the array and then insert values into it. When pre-dimensioning arrays, an initial memory allocation will be performed. This prevents LabVIEW from needing to perform additional allocations, which will cause performance degradation. Figure 9.21 illustrates two array-handling routines. Array copying can be as CPU-intensive as string manipulation. Array variables have four bytes for storage of the array dimensions, and a number of bytes equivalent to the size of the dimensions times the storage size of the type.

9.10.3 When to Use Subroutines

Now that we know the benefits and penalties of using threads and are familiar with implications of data type choices, it is time to determine when subroutines should


Replacing the array elements is much more efficient.
100
2
i
100
0

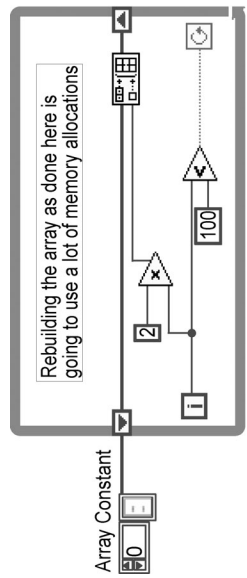

Rebuilding the array as done here is going to use a lot of memory allocations
Array Constant
100
2
i
0

FIGURE 9.21

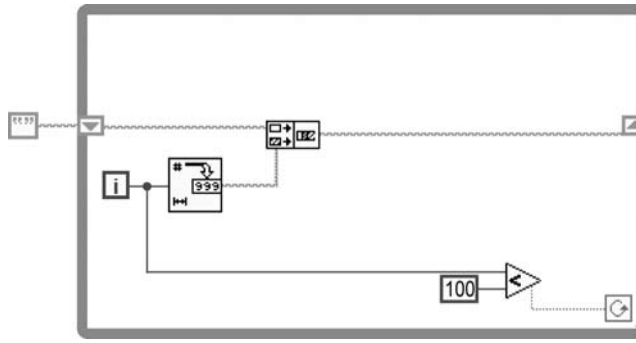


FIGURE 9.22

be used. Knowing that numerical operations are fast and that string and dynamic arrays are slow, the obvious conclusion is that numerical-intensive VIs are prime candidates for subroutine priority.

When handling large strings or variable-dimension arrays, do not use subroutines. When LabVIEW tries to copy strings, the thread performing the copying will incur overhead while the memory allocation is being performed. If this thread is running in a subroutine, other threads may become blocked waiting for one thread to cycle through the subroutine. It is preferable in these situations to have multiple threads working in a VI to minimize blocking points.

Numerical processing is fairly fast; it is possible for threads to completely execute most subroutines in a single timeslice. With modern processors running multiple cores, handling processor instructions out of order, and doing it all with clock rates in the gigahertz, millions of numerical calculations can be accomplished in a single timeslice on a single thread.

As an example of this, we will consider the following problem: the VI depicted in Figure 9.22 shows a simple string-generation routine. The counter is converted to a string and then concatenated to another string and fed through the shift register. Consider the copying and memory allocations that need to be done. Every time the integer is converted to a string, five to six bytes are allocated from heap memory to store the number; five bytes are used when the number is between zero and nine, and six bytes when the number has two digits. Recall that four bytes of length information are stored in a string preamble. The string concatenation requires an additional allocation of length: four bytes + length of old string + length of new string. These allocations are relatively small but add overhead. Figure 9.23 shows the VI profile from numerous runs of this VI.

The timing profile demonstrates that at least one execution required 15 ms to complete. In terms of CPU time, this is significant.

A thread may not have enough time in a quantum to always complete this operation. The thread may then be preempted for higher-priority threads and take some time before it can resume execution. During at least one execution, the thread was preempted and took an order of magnitude more time to complete. Larger string manipulation routines will even take longer. If a single thread is dedicated to per-

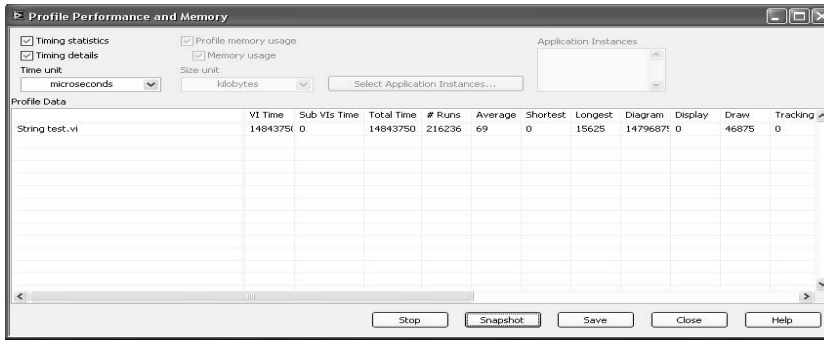


FIGURE 9.23

forming all the manipulations, this could reduce performance of the application. Outputs of this VI will probably be required by other VIs. These other VIs would be blocked from execution while this current VI is completed. The conclusion this example demonstrates is that intensive string manipulations should be performed in VIs that are not subroutines. This will allow multiple threads to perform tasks contained inside the VI. Other threads will not become blocked waiting on a single thread to perform large amounts of memory allocations.

Integer operations require significantly less time to complete, and are often good candidates for subroutine priority. The VI shown in Figure 9.24 shows a simple 100-element manipulation. This type of manipulation may not be common in everyday computing, but it serves as a similar example to the one mentioned above. Figure 9.25 shows the profile window for a large number of runs. Notice the significantly lower timing requirements. It is much less likely that a thread will become blocked during execution of this subVI; therefore, it is desirable to give this subVI subroutine priority because subroutine VIs have less overhead and will execute faster than standard VIs.

When working with arrays in LabVIEW, try to use fixed-length or pre-dimensioned arrays as much as possible. When using the Initialize Array function, one block of memory will be taken from the heap memory. Replacing individual elements in the array will not require the array to be reallocated. Once a fixed array size is defined, then a VI manipulating this array can be a candidate for subroutine priority.

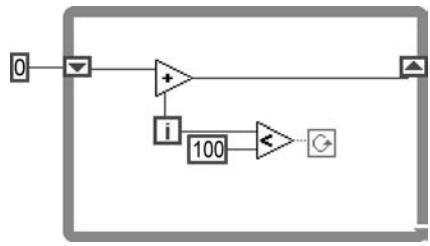


FIGURE 9.24

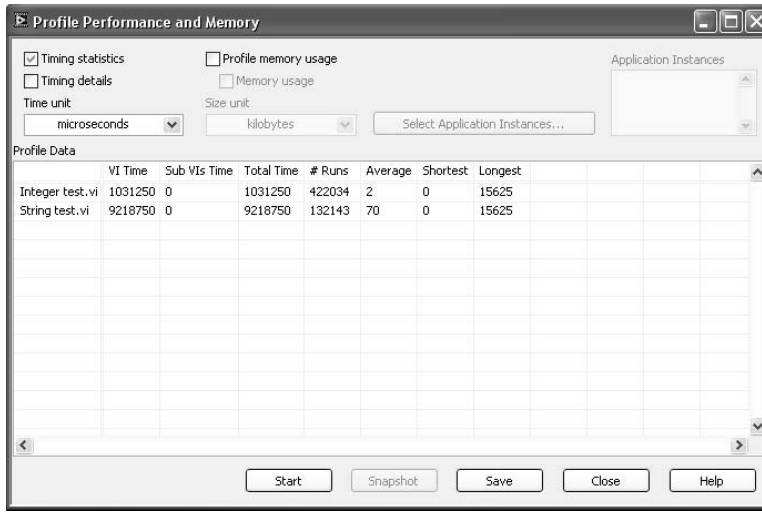


FIGURE 9.25

There will not be significant memory allocations that need to be performed. The overhead on the VI will be reduced, improving the performance of the VI.

9.11 SUMMARY

This chapter began with core multithreading terminology. Threads, processes, applications, and several operating systems were explained. The basics of multithreading — scheduling, priorities, processes, and thread basics — were discussed and defined.

Major myths involving multithreading were addressed. It should never be assumed that threads will improve application performance. Many data flow applications force a serial order of execution; this is precisely where multithreading will be of the least benefit. Another common misunderstanding regarding threads is the idea that the application performance is proportional to the number of threads running. This is only true if multiple processors are available. Rotating threads of execution in and out of the CPU will cause more performance problems than solutions.

Estimating the optimum number of threads is challenging, but not entirely impossible. The programmer must identify where the maximum number of executable elements is generated in the code. Using this number as the maximum number of useful threads will prevent performance limitations.

Subroutine priority VIs can lead to performance gains.

BIBLIOGRAPHY

Microsoft Press, *Windows Architecture for Developers Training*. Redmond: Microsoft Press, 1998.

Aeleen Frisch. *Essential System Administration*, 2nd ed. Cambridge: O'Reilly, 1995.
Bradford Nichols, Dick Buttler, and Jacqueline Prowly Farrel. *Pthreads*. Cambridge: O'Reilly, 1996.
Aaron Cohen and Mike Woodring, *Win32 Multithreaded Programming*. Cambridge: O'Reilly, 1998.
http://www.intel.com/technology/hyperthread/.
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnucmg/html/UCMGch02.asp.

10 Object-Oriented Programming in LabVIEW

This chapter applies a different programming paradigm to G: Object-Oriented Programming (OOP). New languages like Java and its use on the Internet have created a lot of interest in this programming paradigm. This chapter explains the concepts that make object-oriented programming work, and applies them to programming in LabVIEW.

This chapter begins with definitions of objects and classes. These are the fundamental building blocks of OOP. Key definitions that define OOP are then presented which give a foundation for programmers to view applications in terms of their constituent objects.

Once the basics of OOP are described, the first stage of objects is presented — object analysis. Fundamentally, the beginning of the design is to identify the objects of the system. Section 10.4 discusses object design, the process by which methods and properties are specified. The interaction of objects is also defined in the design phase. The third and last phase is the object programming phase. This is where the code to implement the methods and properties is performed.

This type of structuring seems foreign or even backward to many programmers with experience in structured languages such as LabVIEW. Object-oriented is how programming is currently being taught to computer science and engineering students around the world. A significant amount of effort has been put into the design of a process to produce high-quality software. This section introduces this type of philosophy to LabVIEW graphical programming.

Object-oriented design is supported by a number of languages, including C++ and Java. This book tries to refrain from using rules used specifically by any particular language. The concept of object-oriented coding brings some powerful new design tools, which will be of use to the LabVIEW developer. The concept of the VI has already taught LabVIEW programmers to develop applications modularly. This chapter will expand on modular software development.

This chapter discusses the basic methodology of object coding, and also discusses a development process to use. Many LabVIEW programmers have backgrounds in science and engineering disciplines other than software engineering. The world of software engineering has placed significant emphasis into developing basic

design processes for large software projects. The intent of the process is to improve software quality and reduce the amount of time it takes to produce the final product. Team development environments are also addressed in this methodology.

As stated in the previous paragraph, this chapter provides only a primer on object design methodology. There are numerous books on this topic, and readers who decide to use this methodology may want to consult additional resources.

10.1 WHAT IS OBJECT-ORIENTED?

Object-oriented is a design methodology. In short, object-oriented programming revolves around a simple perspective: divide the elements of a programming problem into components. This section defines the three key properties of object-oriented: encapsulation, inheritance, and polymorphism. These three properties are used to resolve a number of problems that have been experienced with structured languages such as C.

It will be shown that LabVIEW is not an object-oriented language. This is a limitation to how much object-oriented programming can be done in LabVIEW, but the paradigm is highly useful and it will be demonstrated that many benefits of object-oriented design can be used successfully in LabVIEW. This chapter will develop a simple representation for classes and objects that can be used in LabVIEW application development.

10.1.1 The Class

Before we can explain the properties of an object-oriented environment, the basic definition of an object must be explained. The core of object-oriented environments is the "class." Many programmers not familiar with object-oriented programming might think the terms "class" and "object" are interchangeable. They are not. A "class" is the core definition of some entity in a program. Classes that might exist in LabVIEW applications include test instrument classes, signal classes, or even digital filters. When performing object programming, the class is a definition or template for the objects. You create objects when programming; the objects are created from their class template. A simple example of a class/object relationship is that a book is a class; similarly, *LabVIEW Advanced Programming Techniques* is an object of the type "book." Your library does not have any book classes on its shelves; rather, it has many instances of book classes. An object is often referred to as an instance of the class. We will provide much more information on classes and objects later in this chapter. For now, a simple definition of classes and objects is required to properly define the principles of object-oriented languages.

A class object has a list of actions or tasks it performs. The tasks objects perform are referred to as "methods." A method is basically a function that is owned by the class object. Generally speaking, a method for a class can be called only by an instance of the class, an object. Methods will be discussed in more detail in Section 10.2.1.

The object must also have internal data to manipulate. Data that are specified in the class template are referred to as "properties." Methods and properties should

be familiar terms now; we heard about both of those items in Chapter 7, ActiveX. Active X is built on object-oriented principals and uses the terminology extensively.

Experienced C++ programmers know the static keyword can be used to work around the restriction that objects must exist to use methods and properties. The implementation of objects and classes in this chapter will not strictly follow any particular implementations in languages. We will follow the basic guidelines spelled out in many object-oriented books. Rules regarding objects and classes in languages like C++ and Java are implementations of object-oriented theory. When developing objects for non-object-oriented languages, it will be helpful to not strictly model the objects after any particular implementation.

LabVIEW does not have a built-in class object. Some programmers might suspect that a cluster would be a class template. A cluster is similar to a structure in C. It does not directly support methods or properties, and is therefore not a class object. We will use clusters in the development of class objects in this chapter. One major problem with clusters is that data is not protected from access, which leads us to our next object-oriented principal, encapsulation.

10.1.2 Encapsulation

Encapsulation, or data hiding, is the ability for an object to prevent manipulation of its data by external agents in unknown ways. Global variables in languages like C and LabVIEW have caused numerous problems in very large-scale applications. Troubleshooting applications with many global variables that are altered and used by many different functions is difficult, at best. Object-programming prevents and resolves this problem by encapsulating data. Data that is encapsulated and otherwise inaccessible to outside functions is referred to as "private data." Data that is accessible to external functions is referred to as "public data."

The object-oriented solution to the problem of excessive access to data is to make most data private to objects. The object itself may alter only private data. To modify data private to an object, you must call a function, referred to as a method, that the object has declared public (available to other objects). The solution that is provided is that private data may be altered only by known methods. The object that owns the data is "aware" that the data is being altered. The public function may change other internal data in response to the function call. Figure 10.1 demonstrates the concept of encapsulated data.

Any object may alter data that is declared public. This is potentially dangerous programming and is generally avoided by many programmers. As public data may be altered at any time by any object, the variable is nearly as unprotected as a global variable. It cannot be stressed enough that defensive programming is a valuable technique when larger scale applications are being written. One goal of this section is to convince programmers that global data is dangerous. If you choose not to pursue object-oriented techniques, you should at least gather a few ideas on how to limit access to and understand the danger of global data.

A language that does not support some method for encapsulation is not object-oriented. Although LabVIEW itself is not object-oriented, objects can be developed to support encapsulation. Encapsulation is extremely useful in large-scale LabVIEW

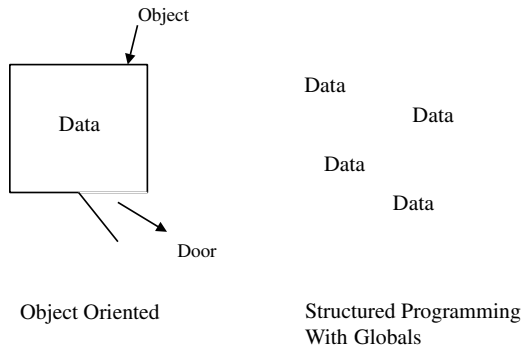


FIGURE 10.1

applications, particularly when an application is being developed in a team environment. Global data should be considered hazardous in team environments. It is often difficult to know which team member's code has accessed global variables. In addition to having multiple points where the data is altered, it can be difficult to know the reason for altering the data. Using good descriptions of logic (DOL) has minimized many problems associated with globals. Using encapsulation, programmers would have to change the variable through a subVI; this subVI would alter variables in a known fashion, every time. For debugging purposes, the subVI could also be programmed to remember the call chain of subVIs that called it.

Encapsulation encourages defensive programming. This is an important mindset when developing large-scale applications, or when a team develops applications. Application variables should be divided into groups that own and control the objects. A small degree of paranoia is applied, and the result is usually an easier to maintain, higher quality application. Global variables have been the bane of many college professors for years. This mindset is important in languages like C and C++; LabVIEW is another environment that should approach globals with a healthy degree of paranoia.

10.1.3 Aggregation

Objects can be related to each other in one of two relationships: "is a" and "has a." A "has a" relationship is called *aggregation*. For example, "a computer has a CD-ROM drive" is an aggregated relationship. The computer is not specialized by the CD-ROM, and the CD-ROM is not interchangeable with the computer itself. Aggregation is a fundamental relationship in object design. We will see later in the chapter that an aggregated object is a property of the owning object.

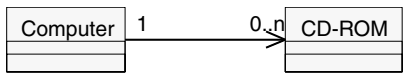


FIGURE 10.2

Aggregation is a useful mechanism to develop complex objects. In an object diagram, boxes represent classes, and aggregation is shown as an arrow connecting the two objects. The relationship between the computer and CD-ROM is shown in Figure 10.2.

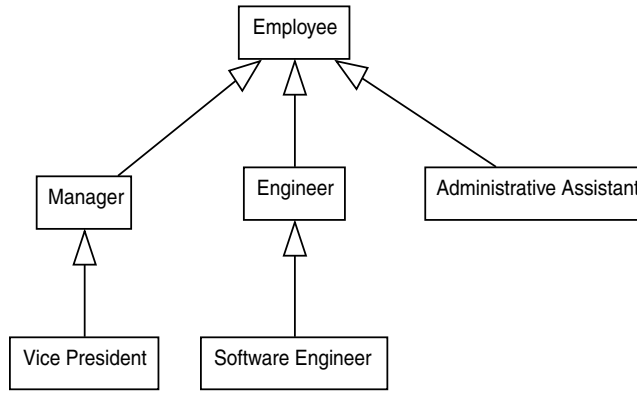


FIGURE 10.3

10.1.4 Inheritance

Inheritance is the ability for one class to specialize another. A simple example of inheritance is that a software engineer is a specialization of an engineer. An engineer is a specialization of an employee. Figure 10.3 shows a diagram that demonstrates the hierarchy of classes that are derived from an employee class. Common in object-oriented introductions is the "is a" relationship. A class inherits from another if it is a specialization or is a type of the superclass. This is a fundamental question that needs to be asked when considering if one class is a specialization of another. Examples of this relationship are engineer "is a" employee, and power supply "is a" GPIB instrument.

When one class inherits from another, the definition of the class is transferred to the lower class. The class that is inherited from is the "superclass" and the inheriting class is the "subclass." For example, consider a class employee. An engineer "is a" employee (please excuse the bad grammar, but it's difficult to be grammatically correct and illustrate an "is a" relationship!). This means that the definition of an employee is used by and expanded by the engineer class. An engineer is a specialization of employee. Other specializations may be vice-president, human resources personnel, and manager. Everything that defines an employee will be used by the subclasses. All employees have a salary; therefore, engineers have salaries. The salary is a property of employee and is used by all subclasses of employee.

All employees leave at the end of the day, some later than others. The function of leaving is common, and is a function that all employee subclasses must use. This method is directly inherited; the same leave function may be used by engineers, vice presidents, and marketing subclasses.

All employees work. Engineers perform different jobs than human resource employees. A definition in the employee class should exist because all employees do some kind of work, but the specifics of the function vary by class. This type of function is part of the employee specification of the employee class but must be done differently in each of the subclasses. In C++, this is referred to as a "pure

virtual function." When a class has a pure virtual function, it is referred to as an "abstract class." Abstract classes cannot be created; only their subclasses may be created. This is not a limitation. In this example, you do not hire employees; you hire specific types of employees.

There is another manner in which classes acquire functions. If employee has a method for taking breaks, and normal breaks are 15 minutes, then most subclasses will inherit a function from employee that lets them take a 15-minute break. Vice presidents take half-hour breaks. The solution to implementing this method is to have a pure virtual method in employee, and have each subclass implement the break function. Object programming has virtual functions. The employee class will have a 15-minute break function declared virtual. When using subclasses such as engineer and the break function is called, it will go to its superclass and execute the break function. The vice president class will have its own version of the break function. When the vice president class calls the break function, it will use its own version. This allows for you to write a single function that many of the subclasses will use in one place. The few functions that need to use a customized version can do so without forcing a rewrite of the same code in multiple places.

Inheritance is one of the most important aspects of object-oriented programming. If a language cannot support inheritance, it is not object-oriented. LabVIEW is not an object-oriented language, but we will explore how many of the benefits of this programming paradigm can be supported in LabVIEW.

10.1.5 Polymorphism

Polymorphism is the ability for objects to behave appropriately. This stems from the use of pointers and references in languages like C++ and Java (Java does not support pointers). It is possible in C++ to have a pointer to an employee class and have the object pointed to be an engineer class. When the work method of the pointer is called, the engineer's work method is used. This is polymorphism; this property is useful in large systems where collections of objects of different type are used.

LabVIEW does not support inheritance, and cannot support polymorphism. We will show later in this chapter how many of the benefits of object-oriented programming can be used in LabVIEW, despite its lack of support for object-oriented programming. Polymorphism will not be used in our object implementation in this chapter. It is possible to develop an object implementation that would support inheritance and polymorphism, but we will not pursue it in this chapter.

10.2 OBJECTS AND CLASSES

The concept of OOP revolves around the idea of looking at a programming problem in terms of the components that make up the system. This is a natural perspective in applications involving simulation, test instruments, and data acquisition (DAQ). When writing a test application, each instrument is an object in the system along with the device under test (DUT). When performing simulations, each element being simulated can be considered one or more classes. Recall from Section 10.1.1 that

an instance of a class is an object. Each instrument in a test rack is an instance of a test instrument class or subclass.

10.2.1 Methods

Methods are functions; these functions belong to the class. In LabVIEW, methods will be written as subVIs in our implementation. The term "method" is not indigenous to object-oriented software, but recall from Chapter 7, ActiveX, that ActiveX controls use methods. Methods may be encapsulated into a class. Methods are considered private when only an instance of the class may use the method.

Methods that are public are available for any other object to call. Public methods allow the rest of the program to instruct an object to perform an action. Examples of public methods that objects should support are Get and Set functions. Get and Set functions allow an external object to get a copy of an internal variable, or ask the object to change one of its internal variables. The Get functions will return a copy of the internal data; this would prevent an external object from accidentally altering the variable, causing problems for the owning object later. Public methods define the interface that an object exposes to other elements of the program. The use of defensive programming is taken to individual objects in object-oriented programming. Only public methods may be invoked, which allows objects to protect internal data and methods.

Only the object that owns itself may call private methods. These types of functions are used to manipulate internal data in a manner that could be dangerous to software quality if any object could alter the internal data. As an example of using objects in a simulation system, consider a LabVIEW application used to simulate a cellular phone network. A class phone has methods to register to the system, make call, and hang up. These methods are public so the program can tell phone objects to perform those actions. Each method, in turn, calls a Transmit method that sends data specific to registration, call setup, or call teardown. The information for each type of message is stored in the specific methods and is passed to the Transmit function. The Transmit function is private to the object; it is undesirable for any other part of the program to tell a phone to transmit arbitrary information. Only specific message types will be sent by the phones. The transmit method may be a common use function internal to the class.

10.2.1.1 Special Method — Constructor

Every class requires two special methods. The first is the Constructor. The Constructor is called whenever an instance of the class is created. The purpose of the Constructor is to properly initialize a new object. Constructors can effectively do nothing, or can be very elaborate functions. As an example, a test instrument class for GPIB instruments would have to know their GPIB address. The application may also need to know which GPIB board they are being used on. When a test instrument object is instantiated, this information is passed to the function in the Constructor. This allows for the test instrument object to be initialized when it is created, requiring no additional configuration on the part of the programmer. Constructors are useful

when uninitialized objects can cause problems. For example, if a test instrument object ends up with default GPIB address of 0 and you send a message to this instrument, it goes back to the system controller. In Section 10.7.1 we will implement Constructor functions in LabVIEW.

The Constructor method is something that cannot be done with simple clusters. Clusters can have default values, but a VI to wrap around the cluster to provide initialization will be necessary. The Constructor function in LabVIEW will be discussed in Section 10.7. Initialization will allow an object to put internal data into a known state before the object becomes used. Default values could be used for primitive data types such as integers and strings, but what if the object contains data that is not a primitive type, such as a VISA handle, TCP handle, or another object? Constructors allow us to set all internal data into a known state.

10.2.1.2 Special Method — Destructor

The purpose of the Destructor is the opposite of the Constructor. This is the second special method of all classes. When an object is deleted, this function gets called to perform cleanup operations such as freeing heap memory. LabVIEW programmers are not concerned with heap memory, but there are cases when LabVIEW objects will want to have a Destructor function. For instance, if when an object is destroyed it is desirable to write information on this object to a log file. If a TCP conversation were encapsulated into a class object, the class Destructor may be responsible for closing the TCP connection and destroying the handle.

In languages such as C++, it is possible to have an object that does not have a defined Constructor or Destructor. The compiler actually provides default functions for objects that do not define their own Constructor and Destructor. Our implementation does not have a compiler that will graciously provide functions that we are too lazy to write on our own. The object implementation presented later in this chapter requires Constructors for all classes, but Destructors will be optional. This is not usually considered good programming practice in object-oriented programming, but our implementation will not support the full features of OOP.

10.2.2 Properties

Properties are the object-oriented name for variables. The variables that are part of a class belong to that class. Properties can be primitive types such as Booleans, or can be complex types such as other classes. Encapsulating a class inside of another class is aggregation. We will discuss aggregation again later in this chapter. An example of a class with class properties is a *bookshelf*. The bookshelf itself is a class with an integer property representing the number of shelves. If the shelf were defined to have a single book on the "A" shelf, then a property to describe the book would be necessary. The description of the book is defined as a class with its own properties, such as number of pages.

Properties defined for a class need to have some relevance to the problem to be solved. If your shelf class had a color constant to represent the color of the shelf, this information should be used somewhere in the program. Future considerations

are acceptable; for instance, we do not use the color information now, but the next revision will definitely need it. If extra properties are primitive types, such as Booleans, there will not be any significant problems. When extra properties are complex types or use resources such as TCP conversations, performance issues could be created because of the extra resources the classes use in the system.

The values of an object's properties make the object unique. All objects of a class have the same methods and property types. Differentiation between objects can be done only with the property values. An example of this would be a generic GPIB instrument class. This class may have properties such as GPIB board and GPIB address. The values of board and address make different GPIB instruments unique. All GPIB objects would have the same methods and property types (address and board number). The value of the address and board make the particular GPIB object unique.

Most properties are private to the class. This means that only the class itself may modify the member variables (properties). This is another measure of defensive programming. Global data has caused countless headaches for C programmers, and encapsulation is one solution to preventing this problem in object-oriented applications. The implementation for objects in this chapter will effectively make all properties private. This means that we will have to supply methods for modifying data from outside the class.

10.3 OBJECT ANALYSIS

Object analysis is the first stage in an object-oriented design process. The objects that comprise the system are identified. The object analysis phase is the shortest phase, but is the most important. Practical experience has shown us that when the object analysis is done well, many mistakes made in the design and program phases have reduced impacts. Good selection of objects will make the design phase easier. Your ability to visualize how objects interact will help you define the needed properties and methods.

When selecting objects, every significant aspect of an application must be represented in one of the objects. Caution must be exercised to not make too many or negligible-purpose objects. For example, when attempting to perform an object analysis on a test application using GPIB instruments, an object to represent the GPIB cables will not be very useful. As none of the GPIB interfaces need to know about the cables, encapsulating a cable description into an object will not be of any benefit to the program. No effort is being made at this point to implement the objects; a basic understanding of which objects are necessary is all that should be needed. The design and interactions should be specified well in advance of coding. Having the design finalized allows for easier tracking of scheduling and software metric collection. This also eliminates the possibility of "feature creep," when the definition of what a function is supposed to do keeps getting changed and expanded. Feature creep will drag out schedules, increase the probability of software defects, and can lead to spaghetti code. Unfortunately, spaghetti code written in LabVIEW does have a resemblance to noodles.

TABLE 10.1
Object Analysis Example #1

Equipment	Purpose
Multimeter	Measure DC bias on output signal line to Device under Test(DUT).
Arbitrary waveform generator	Generate test signal stimuli. The signal is generated on the input signal line to the DUT.
Device under test	The Meaningless Object in Example (MOIE).

Example 1:

This example attempts to clarify the purpose of the object analysis on the design of an application to control an Automated Test Equipment (ATE) rack. This example will start out with a relatively simple object analysis. We will be testing Meaningless Objects in Example (MOIE). Table 10.1 identifies the equipment used in the rack. The MOIE has two signal lines, one input line and one output line.

Solution 1:

We will attempt to make everything available an object. Figure 10.4 shows a model of the objects that exist in the rack. This model is not very useful; there are far too many objects from which to build a solution.

The objects for the GPIB cables are not necessary as the test application will not perform any operations directly with the cables. GPIB read and write operations will be performed, but the application does not need to have internal representations for each and every cable in the system. The objects for the input voltages to the test instruments also make for thorough accounting for everything in the system. Nevertheless, if the software is not going to use an object directly, there is no need to account for it in the design.

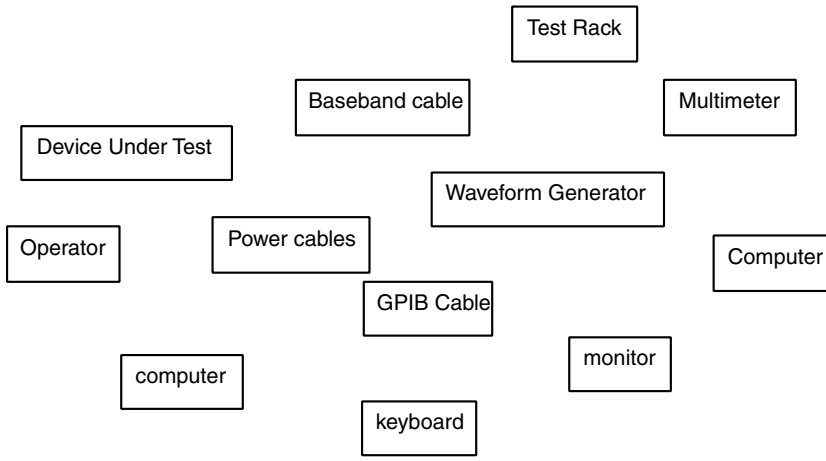


FIGURE 10.4

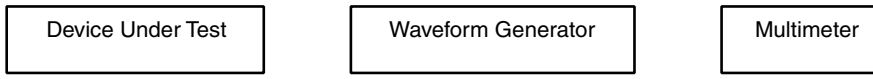


FIGURE 10.5

We are not using objects to represent GPIB cables, but there are times when a software representation of a cable is desirable. If you are working with cables for signal transmission or RF/microwave use, calibration factors may be necessary. An object would be useful because you can encapsulate the losses as a function of cable length, current supplied, or any other relevant factors.

Solution 2:
The easiest object analysis to perform is to declare the three items in the test rack to be their own objects. The simple diagram in Figure 10.5 shows the object relation. This could be an acceptable design, but it is clearly not sophisticated. When performing object analysis, look for common ground between objects in the system. If two or more objects have common functions, then look to make a superclass that has these methods or properties, and have the objects derive from them. This is code reuse in its best form; you need to write the code only once.

Solution 3:
Working from Solution 1 with the three objects (the DUT, waveform generator, and multimeter objects), consider the two GPIB instruments. Both obviously read and write on the GPIB bus; there is common ground between the instruments. A GPIB instrument class could be put into the model, and the meter and waveform generator could inherit the read and write functions of the superclass. Figure 10.6 shows the model with the new superclass.

Which is the best solution, 2 or 3? The answer to that question is: It depends. Personal preferences will come into play. In this example, GPIB control is not necessarily a function that needs to be encapsulated in an object. LabVIEW functions easily supply GPIB functions, and there may be little benefit to abstracting the control to a superclass. The GPIB instrument object needs to be implemented, and this may become more work for the programmer.

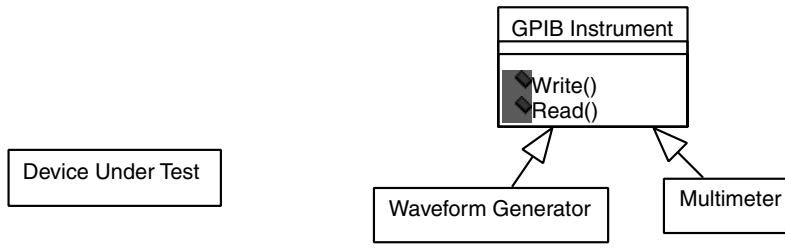


FIGURE 10.6

Example 2:

This example will perform an object analysis on generic test instruments. We will be using the results of this example throughout the chapter. The analysis here will be the focus of a design example in the next section. We have a need to design a hierarchy of classes to support the development of test instrument drivers. An object analysis starting this vaguely needs to start with a base class named "instrument." This base class defines the basic properties and methods that all test instruments would have. Base classes such as this typically have a few properties and methods, and that is about it. Making common ground can be difficult.

A common property would be an address. This address could be used for the GPIB primary address, or a COM port number for a serial instrument. If we define an address to be a 32-bit number, then it could also be the IP address for Ethernet instruments (should they come into existence soon). This is really the only required property in the abstract base class because it is the only common variable to the major communications protocols. For example, we would not want a baud rate property because TCP/IP or GPIB instruments would not use it. Information on the physical cable lengths is not necessary because only high-speed GPIB has any need for this type of information.

All instruments will likely need some type of read and write methods. These methods would be "pure virtual." Pure virtual means that every subclass must support these methods, and the base class will supply no implementation, just the requirement that subclasses have these methods. Pure virtual methods allow each of the base classes to supply custom functionality to the read and write method, which is inherently different for a serial-, GPIB-, or TCP/IP-based instrument. By defining read and write methods, we have effectively standardized the interface for the objects. This is essentially what the VISA library did for LabVIEW; we are repeating this effort in an object-oriented fashion.

The subclasses of instruments for this example will be serial, GPIB, and IP instruments. Obviously, IP is not a common communication protocol for the test industry, but its representation in the object diagram allows for easy future expansion. The GPIB class will cover only the IEEE 488 standard (we will make 488.2 instruments a subclass of GPIB). The following paragraphs will identify the properties and methods that are required for each of the subclasses.

Serial instruments need to implement the read and write methods of the instrument class. Their COM port information will be stored in the address property, which is inherited from instrument. Specific to serial instruments are baud rates and flow control information. These properties will be made private to serial instruments. A Connect method will be supplied for serial instruments. This method will allow for the object to initialize the serial port settings and send a string message if desired by the programmer. The Connect method will not be required in the base class instrument because some instrument subclasses do not require a connection or initialization routine to begin operation — namely, GPIB instruments.

GPIB instruments require a GPIB board and have an optional second address. These two properties are really the only additional items for a GPIB instrument. GPIB instruments do not require a Connect method to configure their communications port. This object must supply Read and Write methods because they derive

from "instrument." Other than read, write, board number, and secondary address, there is little work that needs to be done for GPIB instruments.

As we mentioned previously, we intend to have an IEEE 488.2 class derived from the GPIB instrument class. Functionally, this class will add the ability to send the required commands (such as *RST). In addition to read and write, the required commands are the only members that need to be added to the class. Alert readers will have noticed that we have not added support for high-speed GPIB instruments. Not to worry, this class makes an appearance in the exercises at the end of this chapter.

IP instruments are going to be another abstract base class. This consideration is being made because there are two possible protocols that could be used beneath IP, namely UDP and TCP. We know that both UDP- and TCP-based instruments would require a port number in addition to the address. This is a property that is common to both subclasses, and it should be defined at the higher-level base class. Again, IP will require that UDP and TCP instruments support read and write methods. An additional pair of properties would also be helpful: destination port and address. These properties can again be added to the IP instrument class.

To wrap up the currently fictitious IP instruments branch of the object tree, UDP and TCP instruments need an initialization function. We will call UDP's initialization method "initialize," and TCP's method will be called "connect." We are making a differentiation here because UDP does not maintain a connection and TCP does. TCP instruments must also support a disconnect method. We did not include one in the serial instrument class because, in general, a serial port can effectively just go away. TCP sockets, on the other hand, should be cleaned up when finished with because they will tie up resources on a machine. The object diagram of this example can be seen in Figure 10.7. Resulting from this object analysis is a hierarchy describing the instrument types. This hierarchy can be used as the base for deriving classes for specific types of instruments. For example, a Bitter-2970 power supply may be a particular serial instrument. Serial Instrument would be the base class for this power supply, and its class could be put into the hierarchy beneath the serial instrument. All properties — COM port, methods, read and write — would be supported by the Bitter-2970 power supply, and you would not need to do a significant amount of work to implement the functionality.

Example 3:

This example is going to be a bit more detailed than the previous one. We are going to perform the object analysis for the testing of a Communications Analyzer. This is a compound test instrument that has the functionality of an RF analyzer, RF generator, Audio Analyzer, and Audio Generator. The instrument has a list of other functions that it is capable of doing, but for the purposes of this example we will consider only the functions mentioned.

The first step in the object analysis is to determine what the significant objects are. The RF generator, AF generator, RF analyzer, and AF analyzer are fairly obvious. The HP8920 is compliant with IEEE 488.2, so it has GPIB control with a couple of standard instrument commands. As the instrument is GPIB-based, we will start with abstract classes for instrument and GPIB instrument. A GPIB instrument is a specialization of an instrument. Further specification results in the 488.2 GPIB

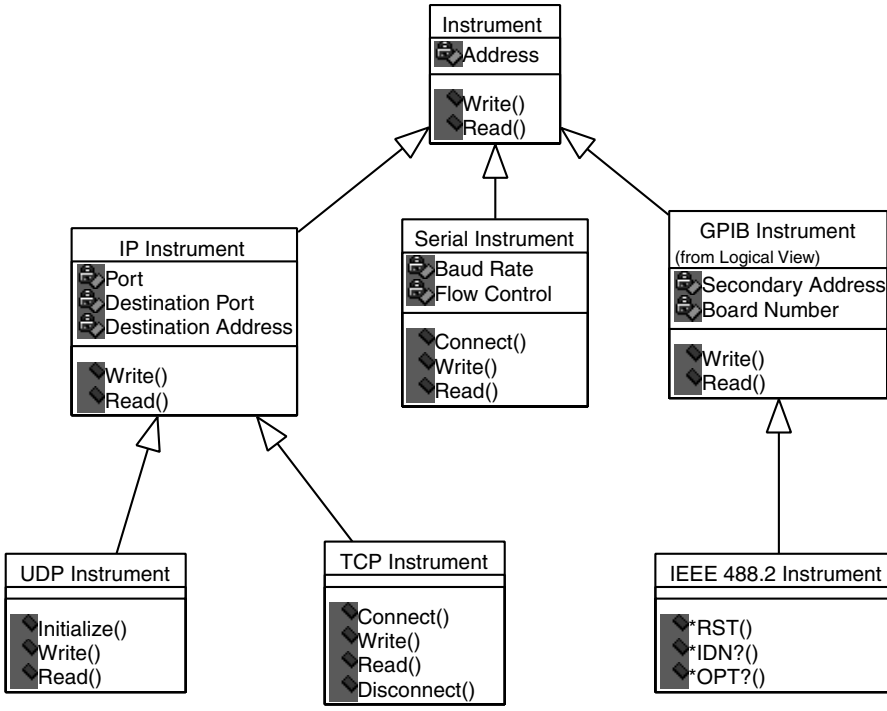


FIGURE 10.7

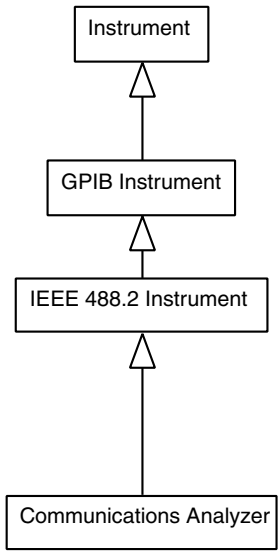


FIGURE 10.8

Instrument subclass. Figure 10.8 shows how our hierarchy is progressing. Some communication analyzers have additional capabilities, including spectrum analyzers, oscilloscopes, and power supplies for devices under test. These objects can also be added to the hierarchy of the design.

An HP-8920 is a communications test set. There are a number of communications analyzers available on the market; for example, the Anritzu 8920 and Motorola 2600 are competitive communications analyzers. All of the instruments have common subsystems, namely the RF analyzers and generators, and the AF analyzers and generators. The preceding sentences suggest that having an HP-8920 as a subclass of a 488.2 instrument is not the best placement. There is a communications analyzer sub-

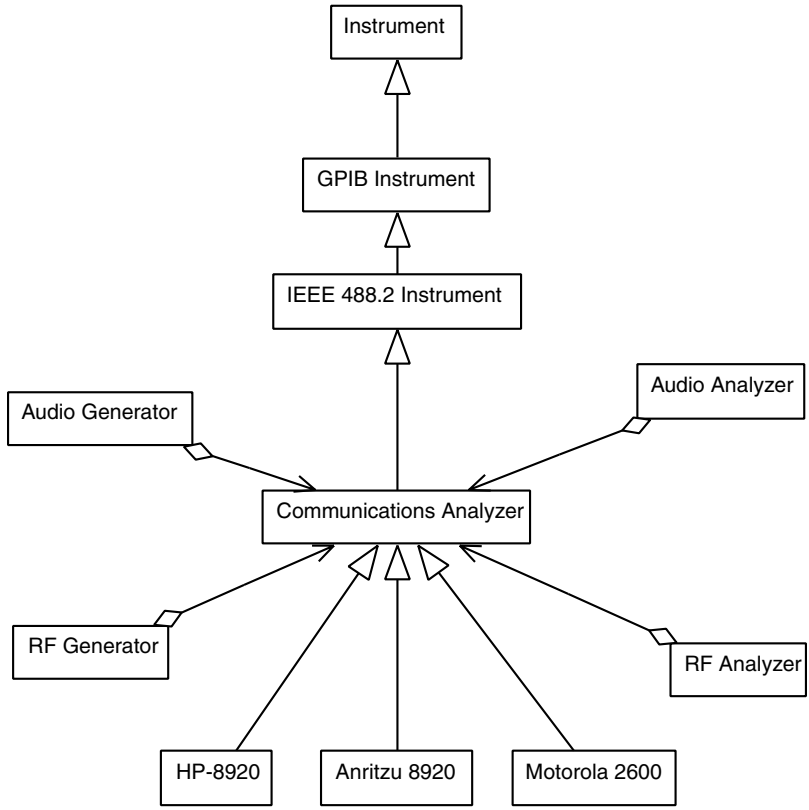


FIGURE 10.9

class of 488.2 instrument. There may also be other subclasses of the 488.2 instrument, such as power supplies. Figure 10.9 shows the expanding hierarchy.

All of the frequency generators have common elements, such as the frequency at which the generator is operating. The RF generator is going to use a much higher frequency, but it is still generating a signal at a specified frequency. This suggests that a generator superclass may be desirable. Our communications analyzer is going to have several embedded classes. Recall that embedded classes are referred to as aggregated classes in object-oriented terminology. We have a new base class to start working with, namely Generator. The AF and RF analyzers have similar starting points, and their analysis is left as an exercise for the reader. Figure 10.10 shows a breakdown of the Analyzer, Generator, and Instrument classes. The dotted line connecting the communications analyzer and the generator/analyzers symbolizes aggregation, encapsulating one class into another. We have a basic design on the core objects. Each class should have a paragraph or so description of the purpose for their design. This would complete an object analysis on this project.

The last objects, oscilloscope, spectrum analyzer, and power supply, are independent objects. There are no base objects necessary to simplify their description. The power supply properties and methods should be fairly easy to decide. The

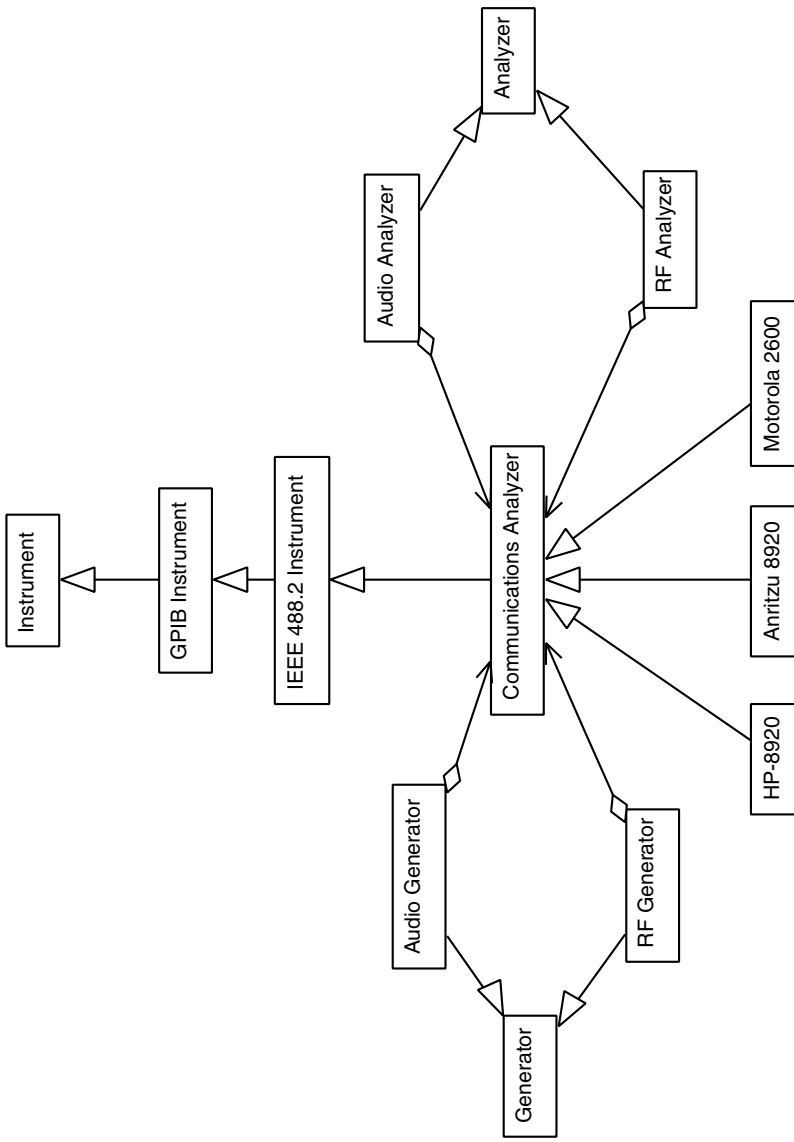

Instrument
GPIB Instrument
IEEE 488.2 Instrument
Communications Analyzer
Audio Analyzer
Analyzer
RF Analyzer
Motorola 2600
Anritzu 8920
HP-8920
Audio Generator
RF Generator
Generator

FIGURE 10.10

oscilloscope and spectrum analyzer are significantly more complex and we are going to leave their analysis and design to the reader.

Object analysis is the first step in object programming, but it is arguably the most important. Good selection of objects makes understanding their interrelations much easier. Now that the basic objects in this system are defined, it is time to focus our attention on how the objects interrelate. The process of analysis and design is iterative; the spiral design model allows you to select your objects, design the objects, and program them. Once you complete one cycle, you can start over and tweak the object list, the function list, and the resulting code.

10.4 OBJECT DESIGN

Object design is where the properties, methods, and interactions of classes are defined. Like object analysis, this phase is pure paperwork. The idea is to have a clearly designed application before code is written. Schedules for coding can be developed with better precision because definitions of functions are well defined. It is much more difficult to estimate code schedules when VIs are defined ad hoc. Coordination of medium to large-scale applications requires project management in addition to a solid game plan. Object designs are well suited for project management processes. Chapter 4, Application Architecture, goes through the details of the waterfall and spiral design models.

A full application design forces architects and programmers to put a significant amount of thought into how the application will be built in the startup phase of a project. Mistakes commonly made during the analysis phase are to neglect or miss application details such as exception handling. Focus at this point of an application should revolve around the interaction of the objects. Implementing individual functions should not be performed unless there are concerns about feasibility.

It is not acceptable to write half the application at this phase of the design unless you are concerned about whether or not a concept is possible. For example, if you were designing a distributed application and had concerns about the speed of DCOM, by all means build a prototype to validate the speed of DCOM. Prototyping is acceptable at this phase because it is much easier to change design decisions when you have not written 100 VIs.

In this section we will explain identifying methods and properties for a class. Once the methods and properties are defined, their interactions can then be considered. The actual implementation of the specified methods and properties will not be considered until the next section. It is important to go through the three distinct phases of development. Once this phase of design is completed, an application's "innards" will be very well defined. The programmers will know what each object is for, how it should function, and what its interactions are with other objects in the system. The idea is to make programming as trivial an exercise as possible. Conceptually, programmers will not have to spend time thinking about interactions of objects because we have already done this. The last phase of the design, programming, should revolve around implementing objects on a method-by-method basis. This allows programmers to focus on the small tasks at hand.

Before we begin a full discussion of object design techniques, we need to expand our class understanding: Sections 10.4.1 and 10.4.2 introduce two concepts to classes. The Container class is useful for storing information that will not be used often. Section 10.4.2 discusses the Abstract class. Abstract classes are useful for defining classes and methods that are used to define interfaces that subclasses must support.

10.4.1 Container Classes

Most, but not all, objects in a system perform actions. In languages such as C++ it is often desirable to encapsulate a group of properties in a class to improve code readability. These container classes are similar in purpose to C structures (also available in C++) and clusters in LabVIEW. Container classes have an advantage over structures and clusters, known as "constructors." The constructor can provide a guaranteed initialization of the container object's properties. Knowing from Chapter 6, "Exception Handling," that the most common problems seen are configuration errors, the object constructor of a container class helps prevent many configuration errors.

Container classes need to support Set and Get functions in addition to any constructors that are needed. If a class performs other actions, then it is not a container class. Container classes should be considered when large amounts of configuration information are needed for objects. Putting this information into an embedded class will improve the readability of the code because the property list will contain a nested class instead of the 20 properties it contains.

10.4.2 Abstract Classes

Some classes exist to force a structure on subclasses. Abstract classes define methods that all subclasses use, but cannot be implemented the same way. An example of an abstract class is an automobile class. A sport utility has a different implementation for Drive than a subcompact car. The automobile class requires that all subclasses define a Drive method. The actual implementation is left up to the subclass. This type of method is referred to as "pure virtual." Pure virtual methods are useful for verifying that a group of subclasses must support common interfaces. When a class defines one or more pure virtual methods, it is an abstract class. Abstract classes may not be instantiated by themselves. An abstract class has at least one method that has no code supporting it.

An abstract class available to the LabVIEW programmer is the Instrument abstract class we designed in Section 10.3. This class requires several methods be supported. Methods to read and write should be required of all instrument derivatives, but serial instruments have different write implementations than GPIB instruments. Therefore, the Write method should be specified as a pure virtual function in the instrument base class. A pure virtual function is defined, but not implemented. This means that we have said this class has a function, but we will not tell you how it works. When a pure virtual function is defined in a class, the class cannot be instantiated into an object. A subclass may be instantiated, but the subclass must also provide implementation for the method.

Continuing on the instrument base class, consider adding a Connect method. Future TCP-based instruments would require a Connect method. Considering future expansions of a class is a good design practice, but a Connect method is not required by all subclasses. Some serial devices may have connect routines, but some will not. GPIB instruments do not require connections to be established. Connections to the VISA subsystem could be done in the constructor. The conclusion is that a Connect method should not be defined in the instrument base class. This method may become a required interface in a subclass; for example, the Connect method can be defined in a TCP Instrument subclass.

The highest class defined in many hierarchies is often an abstract class. Again, the main purpose of the top class in a hierarchy is to define the methods that subclasses must support. Using abstract classes also defines the scope of a class hierarchy. A common mistake made by many object designers is to have too many classes. The base class in a hierarchy defines the scope of that particular class tree. This reduces the possibility of introducing too many classes into a design.

To expand on the object analysis begun in the previous section, consider Example 2. We have the base class "Instrument." All instruments have a primary address, regardless of the communications protocol they use. For a GPIB instrument, the address is the primary address. For a serial instrument, the COM port number can be the primary address. Leaving room for the future possibility of Ethernet and USB instruments, the address property will be a 32-bit number and will be used by all instruments. A 32-bit number was chosen because IP addresses are actually four 8-bit numbers. These 8-bit numbers can be stored in a single 32-bit number. This is actually what the String to IP function does in LabVIEW. Because the address is common to all major subsystems, we will define it as a property of the Instrument base class.

We have identified one property that is common to all instruments; now on to common methods. We know that all instruments must read and write. The Read and Write functions will be different for each type of instrument; therefore, the Instrument class must have two pure virtual methods, Read and Write. Languages like C++ use strong type checking, which means you must also define the arguments to the function and the return types. These arguments and return types must match in the subclass. The good news for us is that we are not required to follow this rule. All instruments so far must read and write strings and possess an address. This seems like a good starting point to the instrument architecture. Figure 10.11 shows a Rational Rose drawing of the Instrument class.

The subclass of Instrument that we will design now is the GPIB instrument subtype. Here we are forced with a decision: which properties does a GPIB instru-

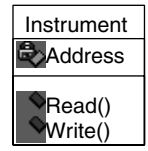


FIGURE 10.11

ment require? Earlier, we decided to use the VISA subsystem for communications. VISA will handle the communications for our GPIB instruments in this example also. The property that a GPIB instrument class requires is a VISA handle. To generate the handle, the primary address, secondary address, and GPIB board must be given in the constructor. The GPIB Instrument class now has one required constructor. Support for read and write must be provided. Read and Write functions are abstract in the Instrument base class, so we must provide the functionality in this class. These methods and properties pretty much encapsulate the functionality of most GPIB (IEEE 488) instruments.

Some instruments, namely IEEE 488.2-compliant instruments have standard commands, such as Reset (*RST), Identity (*IDN?), and Options (*OPT?). Literally, IEEE 488.2 instruments are a subclass of GPIB instruments, and they will be in this object design. The 13 standard commands will be encapsulated as functions in the 488.2 Instrument subclass. The problem that we are developing is that we are in the second phase of the programming, object design. This class should have been thought of during the object analysis. When following a waterfall design methodology, we should stop performing the design and return to the analysis phase. After a moment of thought, several new features for the 488.2 subclass could become useful. For example, the Identification (*IDN?) command could be used in the constructor. This would allow the application to validate that the instrument on the GPIB bus is, in fact, a 488.2 instrument. If the instrument did not respond, or responds incorrectly to the *IDN? command, an error could be generated by the constructor. This type of functionality could be very useful in an application — better user validation without adding a lot to the program.

Now that methods required of Instrument, GPIB Instrument, and IEEE 488.2 Instrument have been defined, it is time to make use of them. Any instruments built using objects will be subtypes of either IEEE 488.2 or GPIB Instrument. If the physical instrument complies to the 488.2 standard, it will be a subclass of the 488.2 Instrument class. In the event we are working with an older, noncompliant instrument, then it will descend directly from GPIB Instrument. This will allow us to make sure that 488.2 commands will never be sent to a non compliant instrument.

As far as defensive programming goes, we have made excellent progress in defending the communications ports. Each instrument class encapsulates the communications port information and does not directly give access to any of the external code. This will make it impossible for arbitrary commands to be sent to any of the instruments. We will limit the commands that are sent to instruments to invoke the objects issue. Assuming the objects are written correctly, correct commands can only be sent on the communications lines.

Another set of methods that you can define for classes is operators. If you had an application that used vector objects, how would you perform addition? An Addition operator that accepts two vector objects can be written. It is possible to specify operators for all the arithmetic operators such as Addition, Subtraction, Multiplication, and Division. Also, Equality operators can be defined as methods that a class can support. It may be possible to use LabVIEW's Equality operator to directly compare the string handles to our objects, but there are some instances where comparing the flattened strings may not yield the results desired.

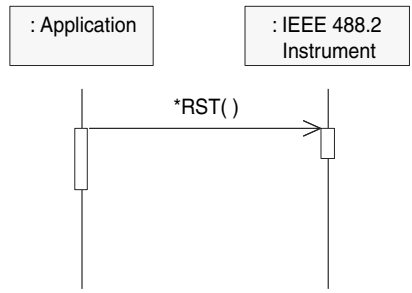


FIGURE 10.12

We have just identified a number of methods that will exist in the Instrument class hierarchy. To help visualize the interaction among these objects, we will use an interaction diagram. Software design tools such as Rational Rose, Microsoft Visual Modeler, and Software through Pictures provide tools to graphically depict the interaction among classes.

As the implementation of classes that we will be using later in this chapter does not support inheritance, we will aggregate the superclasses. The interaction diagram will capture the list of VI calls necessary to accomplish communication between the objects. In an interaction diagram, a class is a box at the top of the diagram. A vertical line going down the page denotes use of that class. Arrows between class objects descend down the vertical lines indicating the order in which the calls are made. Interaction diagrams allow programmers to understand the intended use of classes, methods, and their interactions. As part of an object design, interaction diagrams provide a lot of information that class hierarchies cannot.

If we had an application that created an IEEE 488.2 Instrument object and wanted to send an *RST to the physical instrument, the call chain would be fairly simple. At the upper left of the interaction diagram, an object we have not defined appears — Application. This "class" is a placeholder we are using to indicate that a method call is happening from outside an object. Next in line appears IEEE 488.2 Instrument. The arrow connecting the two classes indicates that we are invoking the *RST method. When the *RST method is called, a string *RST will be created by the object and sent to the GPIB Write function. This encapsulation allows us to control what strings will appear on the GPIB bus. In other words, what we have done is defined a set of function calls that will only allow commands to be sent that are valid. This is effectively an Application Programming Interface (API) that we are defining through this method. The diagram for this interaction appears in Figure 10.12.

This diagram is just one of possibly hundreds for a large-scale application. Interaction diagrams do not clearly define what the methods are expected to do, however; a description of logic (DOL) is required. In addition to interaction diagrams for every possible sequence of function calls, a written description of each call is required. In the DOL should appear function inputs and outputs and a reasonable description of what the function is expected to accomplish. DOLs can be simple, as in the case of the *RST command:

```
*RST-
inputs: error cluster, string handle for object
outputs: error cluster
```

This function sends a string, *RST on the GPIB bus to the instrument defined in the string handle.

For complex functions, the description may become a bit more complicated. Remember that the interaction diagrams are there to help. In a DOL you do not need to spell out an entire call sequence; that is what the interaction diagrams are there for. The DOL and interaction diagrams should leave no ambiguity for the programmers. Together, both pieces of information should minimize the thought process needed to program this particular sequence of events.

The class hierarchy, interaction diagrams, and DOL provide a complete picture for programmers to follow. The interaction diagrams provide graphical direction for programmers to follow in the next phase of development, object programming. Thus far we have managed to define what objects exist in the system, and what methods and properties the objects have. Object interactions are now defined, and each method and property has a paragraph or so describing what it is expected to do. This leaves the small matter of programming the methods and objects. This is not a phase of development for taking shortcuts. Leaving design details out of this phase will cause ambiguities in the programming phase. Any issues or possible interactions that are not covered in the design phase will also cause problems in the programming phase. Software defects become a strong possibility when the design is not complete. Some programmers may neglect any interactions they do not see, and others may resolve the issue on their own. This could well cause "undocumented features," which can cause well-documented complaints from customers. When the software model is complete and up-to-date, it is an excellent resource for how an application behaves. In languages like C++ when an application can have over 100 source code files, it is often easier to look at interaction diagrams and get an idea of where a problem is. Surfing through thousands of lines of source code can be tedious and cause programmers to forget the big picture of the application.

If you do not choose to follow an object-oriented software design methodology, a number of concepts in this chapter still directly apply to your code development. Start with a VI hierarchy and determine what pile of VIs will be necessary to accomplish these tasks. Then write out the interaction diagrams for the possible sequences of events. When writing out interaction diagrams, be sure to include paths that occur when exception handling is in process. Once the interactions are defined, write out descriptions of logic for each of the VIs, and voila! All that is left is to code the individual VIs. Writing the code may not be as easy as just described, but much of the thought process as to how the application should work has been decided. All you have to do is code to the plan.

10.5 OBJECT PROGRAMMING

This section concludes our discussion of the basic process of developing object-oriented code. This is the last phase of the programming, and should be fairly easy

to do. You already know what every needed object is, and you know what all the methods are supposed to do. The challenge is to keep committed to the waterfall model design process. Waterfall design was discussed in Chapter 4, Application Structure. Object programming works well in large-scale applications, and programming technique should be coordinated with a process model for control of development and scheduling.

Once the object analysis and design are done, effort should be made to stick to the design and schedule. In the event that a defect is identified in the design of an application, work on the particular function should be halted. The design should be reviewed with a list of possible solutions. It is important in a large-scale application to understand the effects a design change can have on the entire application. Making design changes at will in the programming phase can cause issues with integration of the application's objects or subsystems. Software quality can become degraded if the impact of design changes is not well understood. If objects' instances are used throughout a program, it should be clearly understood what impact a design change can have on all areas of the application.

In a general sense, there are two techniques that can be followed in assembling an object-oriented application. The first technique is to write one object at a time. The advantage of this technique is that programmers can be dispersed to write and test individual objects. Once each object is written and tested, the objects are integrated into an application. Assuming that all the objects were written and properly tested, the integration should be fairly simple. If this technique is being used, programmers must follow the object design definitions precisely; failure to do so will cause problems during integration.

The second technique in writing an object-oriented application is to write enough code to define the interfaces for each of the objects. This minimally functional set of objects is then integrated into the application. The advantage of this technique is to have the skeleton of the entire application together, which minimizes integration problems. Once the skeleton is together, programmers can fill in the methods and internal functionality of each of the objects. Before embarking on this path, define which need to be partially functional and which need to be fully functional. External interfaces such as GPIB handles may need to be fully functional, whereas report generation code can only be functional enough to compile and generate empty reports.

10.6 DEVELOPING OBJECTS IN LABVIEW

This section begins to apply the previous information to programming in LabVIEW. Our object representations will use clusters as storage containers for the properties of an object. SubVIs will be used as methods. We will develop VIs to function as constructors and destructors, and to perform operations such as comparison. In addition, methods identified in the object design will be implemented.

Cluster type definitions will be used as containers, but we will present only strings to the programmer. Strings will be used as a handle to the object. Clusters will be used for methods because this prevents programmers from "cheating." The idea here is to encapsulate the data and prevent access as a global variable. Pro-

grammers will need to use the Get/Set or other defined methods for access to object "innards." This satisfies the requirement that data be encapsulated. This may seem like a long-winded approach, but it is possible to add functionality to the Set methods that log the VI call chain every time this method is invoked. Knowing which VI modified which variable at which time can be a tremendous benefit when you need to perform some emergency debugging. This is generally not possible if you have a set of global variables that are modified in dozens of locations in the application.

This implementation will not provide direct support for inheritance. Inheritance would require that the flattened string be recognizable as one of a list of possible clusters. Having a pile of clusters to support this functionality is possible, but the parent class should not need to maintain a list of possible child classes. In languages like C++, a virtual function table is used "under the hood" to recognize which methods should be called. Unfortunately, we do not have this luxury. Identifying which method should be called when a virtual method is invoked would be a significant undertaking. This book has a page count limit, and we would certainly exceed it by explaining the logistics of object identification. We will simulate inheritance through aggregation. Aggregation is the ability for one object to contain another as a property. Having the parent class be a property of the child class will simulate inheritance. This technique was used with Visual Basic 4.0 and current versions; Visual Basic does not directly support inheritance. Polymorphism cannot be supported directly because inheritance cannot be directly supported. This is a limitation on how extensive our support for objects can be.

The object analysis has shown us which classes are necessary for an application, and the object design has identified the properties and methods each object has. The first step in implementing the object design is to define the classes. Because LabVIEW is not an object-oriented language, it does not have an internal object representation. The implementation we are developing in this section is not unique. Once you understand the underlying principles behind object-oriented design, you are free to design your own object representations.

10.6.1 Properties

All objects have the same properties and methods. What makes objects of the same type unique are the values of the properties. We will implement our objects with separated properties and methods. The methods must be subVIs, and the class templates for properties will be type definitions using clusters. The cluster is the only primitive data type that can encapsulate a variety of other primitive types, such as integers. Class definitions will be encapsulated inside clusters. The internal representation for properties is clusters, and the external representation will actually be strings. As it is impossible for a programmer to access a member variable without invoking a Get or Set method, our implementation's properties are always private members of the class.

When you program a class definition, the cluster should be saved as a type definition. We will use this type definition internally for all the object's methods. The type definition makes it convenient for us to place the definition in class methods. The cluster definition will be used only in internal methods to the class; programmers

may not see this cluster in code that is not in direct control of the class. External to the object, the cluster type definition will be flattened to a string. There are a number of reasons why this representation is desirable, several of which will be presented in the next section.

Some readers may argue that we should just be passing the cluster itself around rather than flattening it into a string. The problem with passing the class data around as a cluster is that it provides temptation to other programmers to not use the class methods when altering internal data. Flattening this data to a string makes it difficult, although not impossible, for programmers to cheat. Granted, a programmer can always unflatten the cluster from a string and cheat anyway, but at some point we are going to have to make a decision to be reasonable. Flattening the cluster to a string provides a reasonable amount of protection for the internal data of the object.

10.6.2 Constructors

An object design may determine that several constructors will be necessary. The purpose of the constructor is to provide object initialization, and it may be desirable to perform initialization in several different manners. Our class implementation will require that each object have at least one available constructor. The constructor will be responsible for flattening the Typedef cluster into a string as an external handle to the object. Object-oriented languages such as C++ do not require absolutely that each class have a constructor. Like Miranda rights, in C++, if you do not have a constructor, the compiler will appoint one for you. We are supplying an object implementation for LabVIEW, and our first rule is that all objects must have at least one constructor.

A simple example for a class and its constructor is a point. A point has two properties, an x and y coordinate. Figure 10.13 shows the front panel for a constructor function for a point class. The required two inputs, the x and y coordinates, are supplied, and a flattened string representing the points is returned. The code diagram is shown in Figure 10.14. A simple Build Cluster function was used. For more

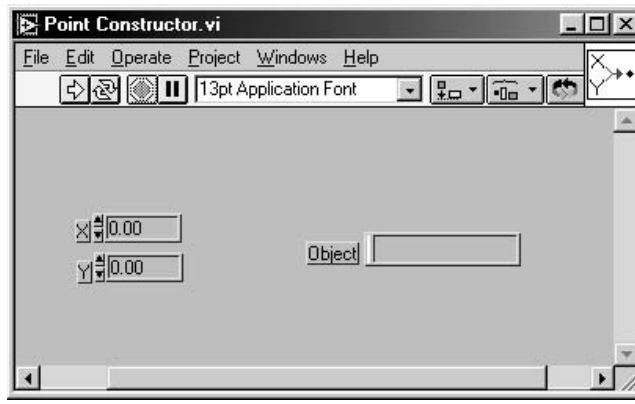


FIGURE 10.13

FIGURE 10.14

complicated functions, you want to build a cluster and save it as a control. This control can be placed on the front panel and not assigned as a connector. This would facilitate the cluster sizing without giving up the internals to external entities.

Another object we will develop using the point is the circle. A circle has a radius and an origin. The circle object will have the properties of a point for origin and a double-precision number for its radius. The cluster control for a circle is shown in Figure 10.15. The circle object constructor, like the point constructor, uses floating-point numbers for radius, x, and y. The front panel is shown in Figure 10.16, and the code diagram is shown in Figure 10.17.

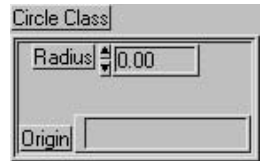


FIGURE 10.15

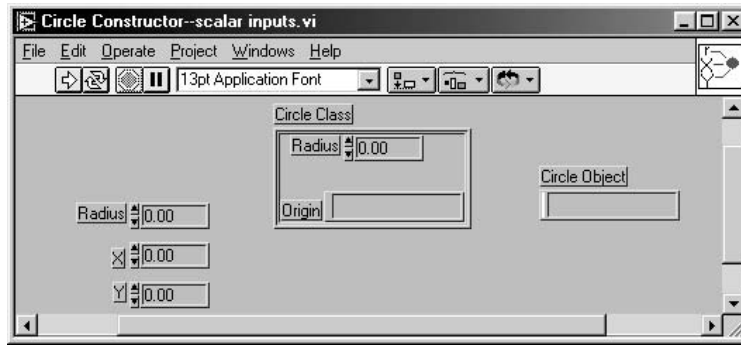


FIGURE 10.16

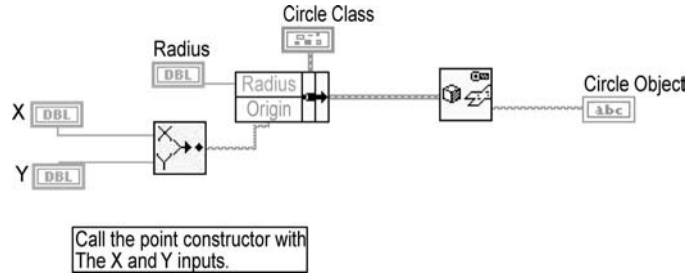


FIGURE 10.17

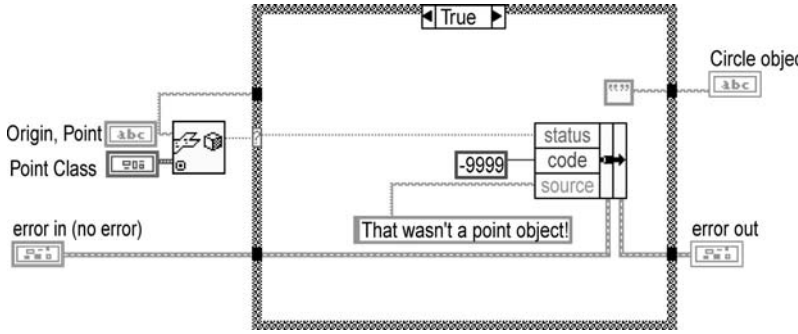


FIGURE 10.18

The front panel uses type definitions for the contents of the clusters. Point coordinates are fed into the point constructor and the string input is fed into the circle's cluster definition. The clusters are used only as inputs to the Bundle function. Next, the cluster is then flattened and returned as a string. The circle uses a nested class as one of its properties.

Now, say that a programmer has the radius and the point itself in a string. Another constructor can be built using the string point object and the radius as inputs. The block diagram for this VI is shown in Figure 10.18. Note that the first thing the constructor does is validate the point object. The error clusters are included in this VI; it is a trivial matter to include them in the other constructors. The circle object demonstrates where multiple constructors are desirable. Functions that have the same purpose, but use different arguments can be called "overloaded." In C++ and Java, a function name can be used many different times as long as the argument list is different for each instance of the function. The function name and argument list is the function's signature in C++; this does not include the return type. As long as different signatures are used, function name reuse is valid. LabVIEW does not have this restriction if desired multiple constructors with the same input list can be used.

Example 10.6.1

Develop a constructor for a GPIB instrument class. All GPIB instruments have primary addresses, secondary addresses, and GPIB boards they are assigned to. All instrument communications will be performed through the VISA subsystem.

Solution:

First, we need to consider the design of the object. The problem statement made it obvious that addresses and GPIB boards are to be used by the object. The statement also says that the communications will be performed via VISA calls. The cluster really needs just one item, a VISA handle. The inputs of addresses and board number can be formatted into a VISA descriptor. The VISA Open VI will open the connection and return the needed VISA handle. Figure 10.19 shows the constructor. Not shown is the front panel, in which the only notable features are the limitations placed on the GPIB address: the primary and secondary addresses must be between 1 and 31. If the secondary address is zero, then we will use the primary address as the secondary address. In addition, the GPIB board number must be between 0 and 7.

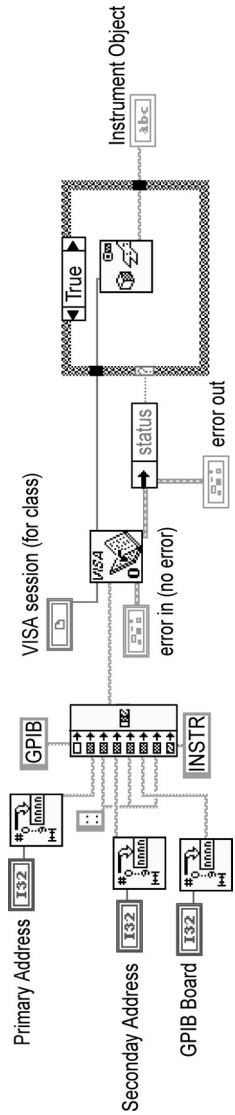

Instrument Object
True
status
error out
VISA session (for class)
error in (no error)
GPIB
INSTR
Primary Address
Seconday Address
GPIB Board
I32
I32
I32

FIGURE 10.19

As this VI will perform communications outside LabVIEW and into the VISA subsystem, error clusters are used to indicate the success or failure of the operation. If VISA Open returns an error, then an empty string is returned as the object in addition to setting the error cluster. Our implementation of objects will return empty strings when objects cannot be created.

10.6.3 Destructors

Destructors in our object implementation need to exist only when some activity for closing the object needs to be done. Any objects used to encapsulate TCP, UDP, VISA, ActiveX (automation), or synchronization objects should destroy those connections or conversations when the object is destroyed. If the object's string is allowed to go out of scope without calling the destructor, LabVIEW's engine will free up the memory from the no-longer-used string. The information holding the references to the open handles will not be freed up. Over long periods of time, this will cause a memory leak and degrade LabVIEW's performance.

Classes such as the point and circle do not require a destructor. The information they hold will be freed up by the system when they are eliminated; no additional handles need to be closed. This is actually consistent with other programming languages such as C++ and Java. Programmers need to implement destructors only when some functionally needs to be added.

Example 10.6.2

Implement a destructor for the GPIB class object created in Example 10.6.1.

Solution:

All that needs to be done for this example is to close the VISA handle. However, when we recover the cluster from the object string, we will verify that it is a legitimate instance of the object. The destructor code diagram is shown in Figure 10.20.

The significance of destructors is important for objects that communicate with the outside world. For internally used objects, such as our point and circle, or for objects simulating things like signals, no destructor is necessary. Unlike constructors, there is only one destructor for an object. In our implementation it is possible to construct multiple destructors, but this should be avoided. It will be obvious in the object design what items need to be closed out; this should all be done in one point. The act of destroying an object will require the use of only one destructor. Having multiple destructors will serve to confuse programmers more than it will help clarify the code.

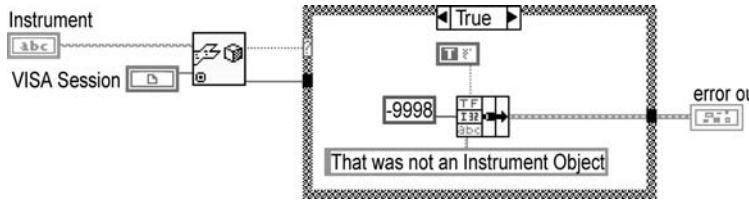


FIGURE 10.20

10.6.4 Methods

In this section we will begin implementing methods that objects will use. The two distinct classifications of methods will be Public and Private methods. Protected methods will not be supported; their implementation will be much more difficult. A stronger class design in the future may allow for protected interfaces. Private and public methods will be handled separately because their interfaces will be different.

Our object implementation will use methods to interface with the outside application exclusively. Setting or retrieving the values of properties will be handled through methods called Set and Get. This is how ActiveX is implemented. Each property that you have read and/or write access to is handled through a function call.

10.6.4.1 Public Methods

Public methods take a reference to the object in the form of a string. This prevents outside code from modifying private properties without a defined interface. A special example of a Public function is the constructor. The constructor takes inputs and returns a string reference to the object. This is a public method because it was called from outside the object and internal properties were configured by a method the object supports.

Class designs may have many public methods, or just a few. Object design tools like Rational Rose generate code that, by default, will include Get and Set functions for access to object properties. We will be using Get and Set functions to alter properties of our objects. The Get/Set functions will form the protective interface for member variables. Set methods will allow us to control or force rules on setting internal data members. This gives us the ability to perform a lot of intelligent coding for the object users. From our previous example, a For GPIB Instrument, we could have a Set GPIB Address method as public. This method allows programmers to allow the object to determine if its GPIB address has been changed during execution. If this is the case, the object could be made capable of refusing to make the change and generate an error, or of closing its current VISA session and creating a new one. Public methods enable the object to make intelligent decisions regarding its actions and how its private data is handled. This is one of the strengths of encapsulation. When designing objects, consideration can be made as to how the Get and Set interfaces will operate. Intelligent objects require less work of programmers who use them because many sanity-checking details can be put into the code, freeing object users from trivial tasks.

10.6.4.2 Private Methods

The major distinction between public and private methods in our LabVIEW implementation is that private methods may use the type definition cluster as input. Private methods are considered internal to the class and may directly impact the private properties of the class. The extra step of passing a string reference is unnecessary; basically, a private method is considered trustable to the class. When using this object implementation in a LabVIEW project, a private method should always appear as a subVI to a public method VI. This will enable you to verify that defensive

programming techniques are being followed. As a quick check, the VI hierarchy can be examined to verify that private methods are only called as subVIs of public methods. Public methods serve as a gateway to the private methods; private methods are generally used to simplify reading the code stored in public methods.

By definition, private methods may be invoked only by the object itself. When can an object call an internal method? Public methods may call private methods. When a public method is executing, it may execute private methods. This may seem to be a waste, but it really is not. This is a good defensive programming practice. The public method is an interface to the external program, and the private methods will be used to accomplish tasks that need to be performed.

As an example of a private method that an object would use, consider implementing a collection of VIs to send e-mail using Simple Mail Transfer Protocol (SMTP). To send mail using SMTP, a TCP connection must be established to a server. The Read and Write TCP functions are not something you would want a user of the SMTP mail object to directly have access to. Implementing the SMTP protocol is done internally to the SMTP object. Again, this is defensive programming. Not allowing generic access to the TCP connection means that the SMTP object has complete control over the connection, and that no other elements of code can write data to the mail server that has not been properly formatted.

Simple objects may not require private methods to accomplish their jobs. A simple object such as a vector does not require a private method to determine its magnitude. In our implementation, private methods are necessary only if they simplify code readability of a public method. There are a number of places where this is desirable. In our GPIB Instrument class, public methods would store the strings needed to send a given command. The common ground between all the public methods would be the need to send a string through the GPIB bus. A Write VI can be made as a private method so you do not need to place GPIB write commands and addressing information in each of the VIs. This can be done in the single private method and dropped into public methods.

10.7 EXAMPLES IN DEVELOPING INSTRUMENT DRIVERS

This section will develop several instrument drivers to illustrate the benefits of object modeling in LabVIEW. This technique works very well with SCPI instruments. SCPI command sets are modularized and easily broken down into class templates. Object-based instruments can be considered alternatives to standard instrument drivers and IVI Instruments. Our first example will concentrate on power supplies, specifically one of the IVI-based instruments.

A simple object model for a power supply would have the core definition of a power supply. It would be fair to assume that GPIB controllers will control a vast majority of power supplies. It would make sense to reuse the core GPIB Instrument class to control GPIB behavior. This design will use a one-class-fits-all approach. In a true object-oriented implementation, it would make sense to have a power supply abstract base class that defines voltage and current limit properties. In addition to the current limit property, we will be supplying a read only property: current. This

will allow users to read the current draw on the supply as a property. As we are adding object implementations to a non-object-oriented language, many of the advantages of abstract base classes do not apply. Instead, we will define the methods and properties that are used by the vast majority of power supplies and implement them with the addition of a model property. This technique will work well for simple instruments, but complex instruments such as oscilloscopes would be extremely difficult. Multiple combinations of commands that individual manufacturer's scopes would require to perform common tasks would be an involving project, which it is for the IVI Foundation.

The purpose of the Model property is to allow users to select from an enumerated list of power supplies that this class supports. Internally, each time a voltage or current is set, a Select VI will be used to choose the appropriate GPIB command for the particular instrument model. From a coding standpoint, this is not the most efficient method to use to implement an instrument driver, but the concepts of the objects are made clear.

The first step in this object design is to determine which properties go inside the cluster Typedef for this class. Obvious choices are values for current limit and voltage. This will allow the power supply driver to provide support for caching of current limit and voltage. No values will be retained for the last current measurement made; we do not want to cache that type of information because current draw is subject to large changes and we do not want to feed old information back to a user. Other values that need to be retained in this class are the strings for the parent classes, GPIB Instrument, and IEEE 488.2 Instrument. The cluster definition is shown in Figure 10.21. The constructor for power supply will call the constructor for IEEE 488.2 Instrument. We need to retain a handle for this object, which will take the GPIB address information as an argument. The GPIB addressing information does not need to be retained in the Typedef for power supply. The IEEE 488.2 Instrument will, in turn, call the constructor for GPIB Instrument, which takes the GPIB address information to its final destination, the VISA handle it is used to generate.

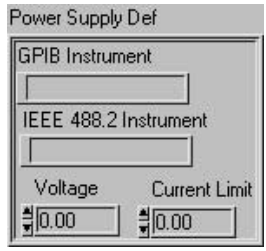


FIGURE 10.21

The Constructor VI for the power supply takes several inputs: the GPIB board number, GPIB primary address, GPIB secondary address, and the power supply model. The power supply constructor will then call the constructor for the IEEE 488.2 Instrument. As we have a higher base class, the IEEE 488.2 Instrument will call the constructor for the GPIB Instrument class. This class will initialize the VISA session and return a string burying this information from users and lower classes. This defensive programming restricts anyone from using this handle in a mechanism that is not directly supported by the base class. Constructor functions for each of the three classes in the chain are shown in Figure 10.22.

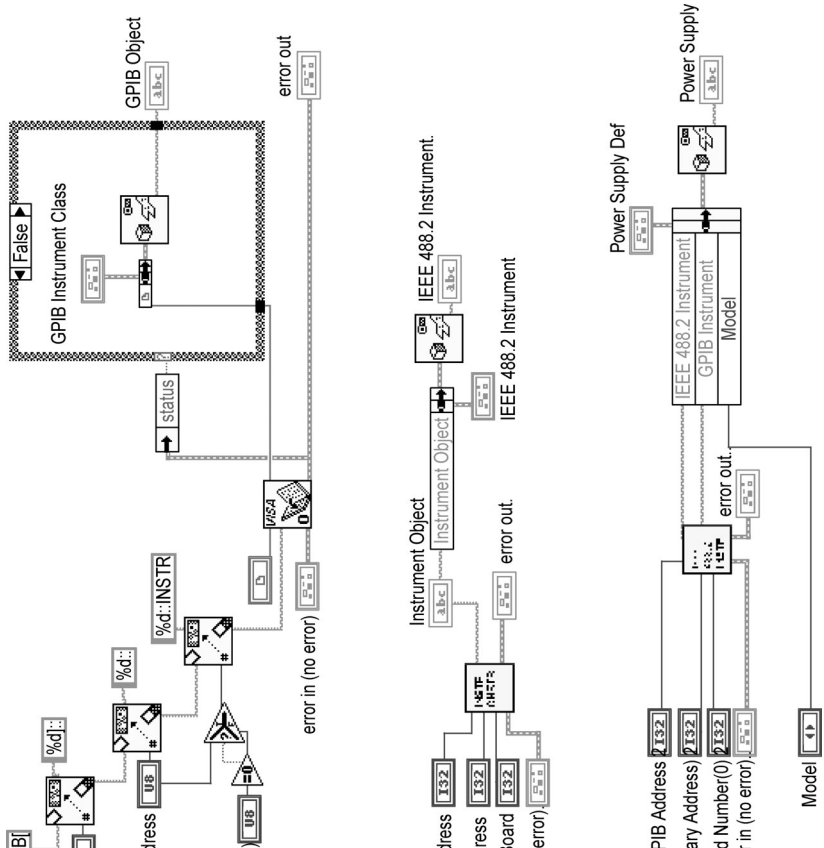

GPIB
GPIB Board Number(0)
%d::
GPIB Address
%d::
Secondary Address (0= Primary Address)
%d::INSTR
False
GPIB Instrument Class
GPIB Object
status
error out
error in (no error)
Primary Address
Seconday Address
GPIB Board
error in (no error)
Instrument Object
error out.
IEEE 488.2 Instrument
Power Supply Def
GPIB Address
Secondary Address (0= Primary Address)
GPIB Board Number(0)
error in (no error)
IEEE 488.2 Instrument
GPIB Instrument
Model
error out.
Power Supply
Model

FIGURE 10.22

The next items that need to be developed are the Get and Set VIs for the properties' voltage and current. Get and Set methods will be supplied for voltage and current limits, but only a Get command needs to be supplied for the current draw property. Current draw is actually a measurement, but from an object design standpoint it is easily considered to be a property. Set Voltage is shown in Figure 10.23. The Get Current Draw method is shown in Figure 10.24.

We are not supplying Get or Set functions for the GPIB address or instrument model information. It is not likely that a user will switch out power supplies or change addressing information at run-time, so we will not support this type of operation. It is possible to supply a Set Address method that would close off the current VISA handle and create a new GPIB Instrument class to reflect the changes. This sounds like an interesting exercise for the reader, and it appears in the exercises at the end of this chapter.

One issue that we face is the lack of inheritance in our classes. The code to support inherited methods is somewhat bulky and limiting. The GPIB Write commands need to propagate from power supply to IEEE 488.2 Instrument to GPIB Instrument. This is a fairly large call chain for an act of writing a string to a GPIB bus. The wrapper VIs do not take a significant amount of time to write, but larger instruments with several hundred commands could require several hundred wrappers, which would take measurable engineering time to write and is a legitimate problem. In true object-oriented languages, the inherited functions are handled in a more elegant fashion.

The two interaction diagrams presented in Figures 10.25 and 10.26 show the sequence of VI calls needed to set the voltage of the power supply. The first diagram shows what happens when the Voltage Set method gives a value equal to the currently-cached voltage. The function simply returns without issuing the command. The second diagram shows that the Voltage Set method had an argument other than the currently-cached command and triggers the writing of a command to the GPIB instrument class.

In the next example, we consider a more complicated instrument. This example will introduce another concept in programming, "Friend classes." Friend classes are the scourge of true object-oriented purists, but they present shortcuts to solving some of our implementation problems.

10.7.1 Complex Instrument Designs

In this example we will implement the communications analyzer we performed the object analysis on in Section 10.3. Now that we understand how our object implementation works in LabVIEW, we can review some of the analysis decisions and move forward with a usable driver.

Some limitations arose in the power supply example we did previously, and we will need to address these limitations. First, power supplies are generally simple instruments on the order of 20 GPIB commands. Modern communications analyzers can have over 300 instrument commands and a dizzying array of screens to select before commands can be executed. The one driver model for a group of instruments will not be easily implemented; in fact, the IVI Foundation has not addressed

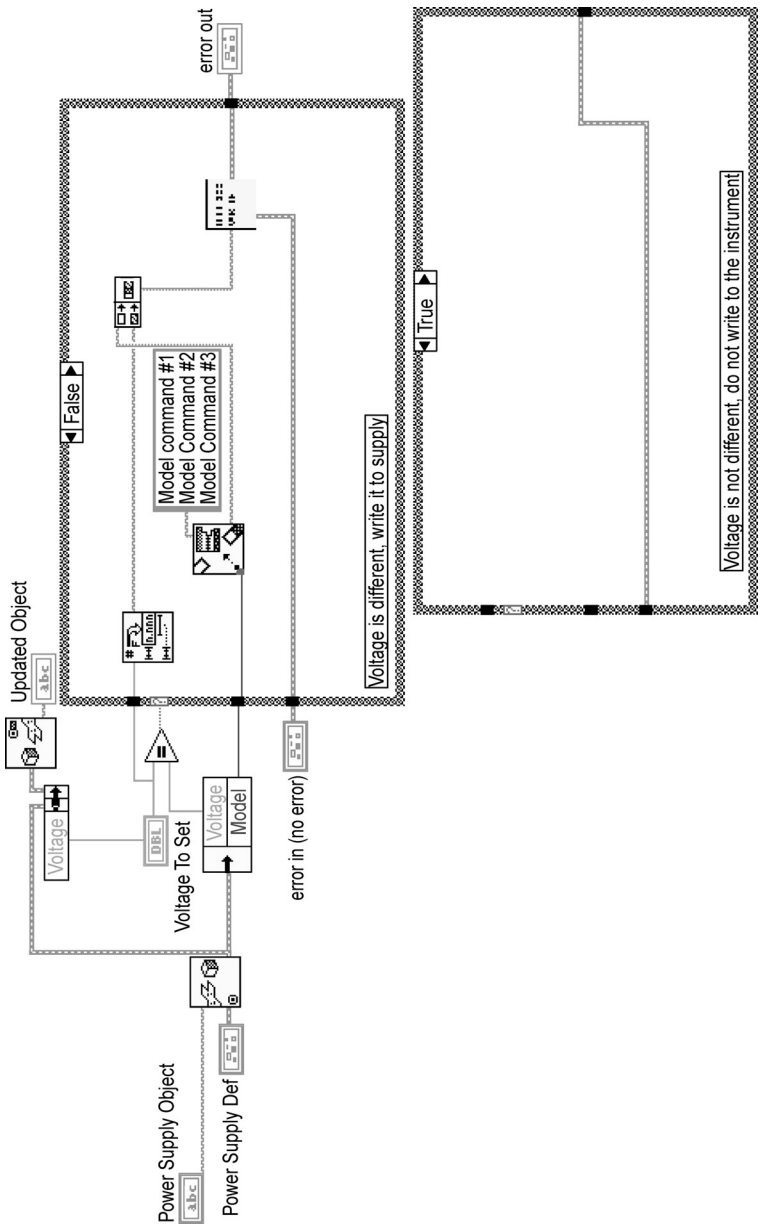

Power Supply Object
Power Supply Def
Voltage
Model
Voltage To Set
DBL
Voltage
Updated Object
error in (no error)
error out
False
Model command #1
Model Command #2
Model Command #3
Voltage is different, write it to supply
True
Voltage is not different, do not write to the instrument

FIGURE 10.23

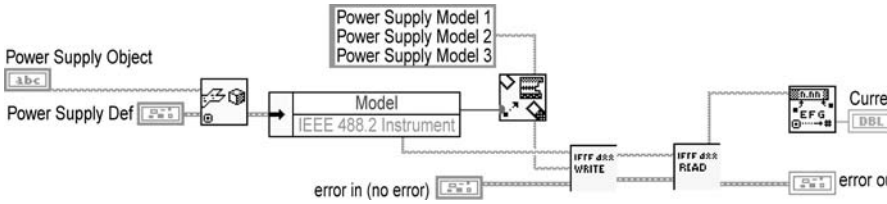


FIGURE 10.24

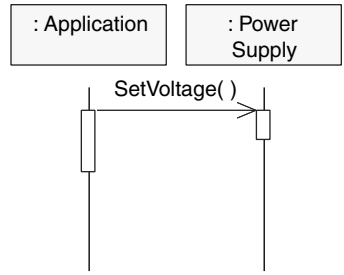


FIGURE 10.25

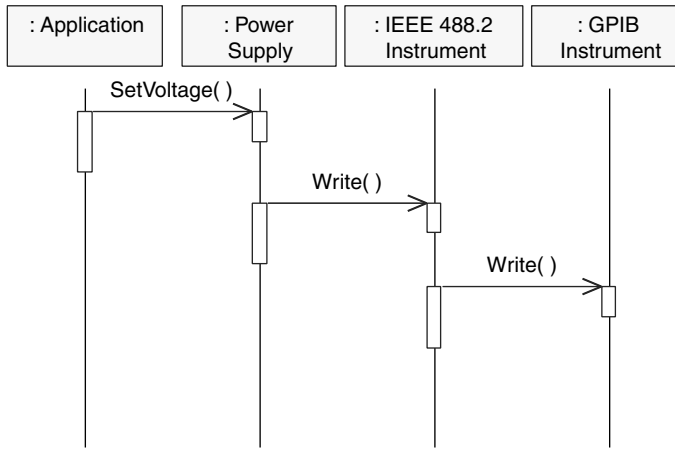


FIGURE 10.26

complex test instruments at this time. The matrix of command combinations to perform similar tasks across different manufacturer's instruments would be difficult to design, and an amazing amount of work to implement. In other words, standardizing complex instruments is a significant undertaking, and it is not one that the industry has completely addressed. It is also not a topic we should address.

The object analysis presented in Section 10.3 works very well in object-oriented languages. Our implementation does have some problems with inheritance, and we will have to simplify the design to make implementation easier. Abstract classes

for the analyzers and generators have a single property and a handful of pure virtual methods. As virtual methods are not usable in our implementation, we will remove them. We will still make use of the IEEE 488.2 class and the GPIB Instrument class. The object hierarchy is greatly simplified. The communications analyzer descends from IEEE 488.2 Instrument, which is a natural choice. The generators, RF and AF, appear on the left side of the communications analyzer. This is an arbitrary choice as there is no special significance to this location. The analyzers appear on the right. The arrow connecting the communications analyzer to the component objects denotes aggregation. Recall that aggregated classes are properties of the owning class; they do not exist without the owning class. This is a natural extension of the object model; this model reflects the real configuration of the instrument. At this point we can claim a milestone in the development: the object analysis is now complete.

Hewlett Packard's HP-8920A communications analyzer will be the instrument that is implemented in this design. We are not endorsing any manufacturer's instruments in this book; its standard instrument driver appears on the LabVIEW CDs and allows us to easily compare the object-based driver to the standard driver.

The next difficulty that we need to address is the interaction between the aggregated components and the communications analyzer. This example will implement a relatively small subset of the HP-8920's entire command set. This command set is approximately 300 commands. Each component object has between 20 and 50 commands. A significant number of wrapper VIs need to be implemented as the aggregated components are private data members. This author is not about to write 300 VIs to prove an example and is not expecting the readers to do the same. We will need to find a better mechanism to expose the aggregate component's functionality to outside the object. The solution to this particular problem is actually simple. As the analyzers and generators are properties of the communications analyzer, we can use Get methods to give programmers direct access to the components. This does not violate any of our object-oriented principals, and eliminates the need for several hundred meaningless VIs. Figure 10.27 shows the list of the properties and methods the communications analyzer has. Properties and methods with a lock symbol next to them are private members and may only be accessed directly by the class itself.

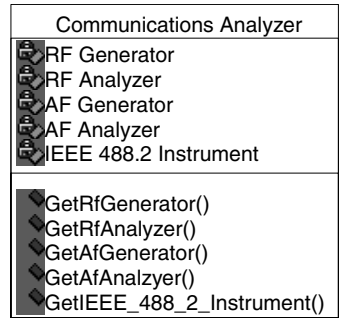


FIGURE 10.27

Now that we have a basic design down for the core object, it is time to examine a small matter of implementation. As we will give programmers access to the analyzers and generators, it stands to reason that the programmers will set properties and expect the components to send GPIB commands to the physical instrument. Reexamining the class hierarchy shows us that we do not have the ability to access the GPIB Instrument class directly from the generator and analyzers because they do not derive from GPIB Instrument. This presents a problem that needs to be resolved before we proceed with the design.

We require that the component objects be capable of accessing the GPIB Read and Write methods of the communications analyzer. It does not make sense to have each of the components derive from the communications analyzer — this does not satisfy the "is a" relationship requirement for subclasses. In other words, the phrase, "an RF generator 'is a' communications analyzer," is not true. Having the component objects derive from the communications analyzer is not a solution to this problem.

We can give the component objects a property for the communications analyzer. Then they can invoke its GPIB Write method. This solution does not realize the class hierarchy we developed in the object analysis phase, either. It is possible to go back and change the class hierarchy, but now we do not satisfy the "has a" requirement for aggregated components. An audio generator "has a" communications analyzer is not a true statement and, therefore, having a property of a communications analyzer does not make much sense.

As none of our solutions thus far make it in our given framework, we will need to expand it somewhat. An RF analyzer is a component in a communications analyzer and it does have access to the onboard controller (more or less). When performing an object analysis and design, it often helps to have the objects interact and behave as the "real" objects do. For the communications analyzer to give access to the GPIB Read and Write methods would violate the encapsulation rule, but it does make sense as it models reality. The keyword for this type of solution in C++ is called "friend." The friend keyword allows another class access to private methods and properties. We have not discussed it until this point because it is a technique that should be used sparingly. Encapsulation is an important concept and choosing to violate it should be justified. Our previous solutions to GPIB Read and Write access did not make much sense in the context of the problem, but this solution fits. In short, we are going to cheat and we have a legitimate reason for doing so. The moral to this example is to understand when violating programming rules makes sense. Keywords such as friend are included in languages like C++ because there are times when it is reasonable to deviate from design methodologies.

Now that we have the mechanism for access to the GPIB board well understood, we can begin implementation of the methods and properties of the component objects. We shall begin with the audio generator. Obvious properties for the audio generator are Frequency, Enabled, and Output Location. We will assume that users of this object will only be interested in generating sinusoids at baseband. Enabled indicates whether or not the generator will be active. Output Location will indicate where the generator will be directing its signal. Choices we will consider are the AM and FM modulators and Audio Out (front panel jacks to direct the signal to an external box). The last property that we need to add to the component is one for

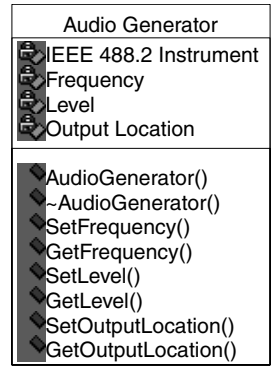


FIGURE 10.28

GPIB Instrument. As the communications analyzer derives from IEEE 488.2 Instrument, we do not have direct access to the GPIB Instrument base class. It will suffice to have the IEEE 488.2 Instrument as a property.

Methods for the audio generator will be primarily Get and Set methods for the Frequency, Enabled, and Output Location properties. Programmers have a need to change and obtain these properties. As this component borrowed the information regarding the GPIB bus, it is not appropriate to provide a Get method to IEEE 488.2 Instrument. A Set method for IEEE 488.2 Instrument does not make much sense either. The audio generator is a component of a communications analyzer and cannot be removed and transferred at will. The IEEE 488.2 Instrument property must be specified at creation as an argument for the constructor. The complete list of properties and methods for the audio generator appear in Figure 10.28.

The two last issues to consider for the audio generator are the abilities to cache and validate its property values. This is certainly possible and desirable. We will require the Set methods to examine the supplied value against that last value supplied. We also need to consider that the values supplied to amplitude are relative to where the generator is routing its output. A voltage is used for the audio out jacks, where FM deviation is used when the signal is applied to the FM modulator. When a user supplies an amplitude value, we need to validate that it is in a range the instrument is capable of supporting. This is not a mission-critical, show-stopping issue; the instrument will limit itself and generate an error, but this error will not be propagated back to the user's code. It makes more sense to have the object validate the inputs and generate an error when inputs are outside of acceptable ranges. RF generator will be subjected to similar requirements.

Last is the destructor for the function. Most of the properties are primitive, floating-point numbers and do not require a destructor. The IEEE 488.2 property has an embedded handle to a VISA session. It would make sense for the control to clean this up before it exits. In this case it is undesirable for the component objects to clean up the VISA handle. A total of five components have access to this property, and only one of them should be capable of destroying it. If we destroy the VISA handle in this object, it will be impossible to signal the other components that the GPIB interface is no longer available. As this component has "borrowed" access to

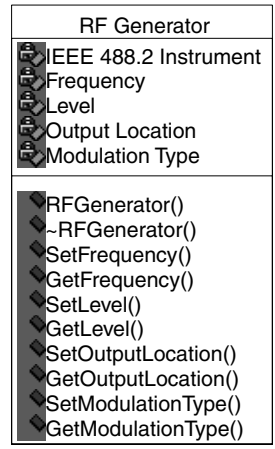


FIGURE 10.29

the GPIB object, it stands to reason that only the communications analyzer object should have the ability to close out the session. Therefore, this object does not need a destructor; everything including the GPIB object property can simply be released without a problem. We must be certain, however, that the destructor for the communications analyzer terminates the VISA session.

The audio and RF generators have a fair amount in common, so it is appropriate to design the RF generator next. The RF generator will need to have an IEEE 488.2 property that is assigned at creation, like the audio generator. In addition, the RF generator will require a Frequency, Amplitude, and Output Port property. Without rewriting the justification and discussion for the audio generator, we will present the object design for the RF generator in Figure 10.29.

Next on the list will be the audio analyzer. This component will not have any cached properties. The Get methods will be returning measurements gathered by the instrument. Caching these types of properties could yield very inaccurate results. The IEEE 488.2 Instrument will be a property that does not have a Get or Set method; it must be specified in the constructor. The audio analyzer will need a property that identifies where the measurement should be taken from. Measurements can be made from the AM and FM demodulator in addition to the audio input jacks. Measurements that are desirable are Distortion, SINAD, and Signal Level (Voltage). The Distortion and SINAD measurements will be made only from the demodulator, whereas the Signal Level measurement can only be made at the audio input jacks. These are requirements for this object.

The object should change the measurement location property when a user requests a measurement that is not appropriate for the current setting. For example, if a user requests the Signal Level property when the measurement location is pointing at the FM demodulator, the location property should be changed to Audio In before the measurement is performed. Measurement location can and should be a cached property. As measurement location is the only cached property, it is the only one of the measurement properties that should be included in this class. SINAD,

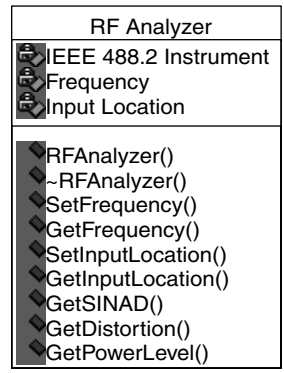


FIGURE 10.30

Distortion, and Signal Level will not be stored internally, and there is no reason why they need to appear in the cluster Typedef. The list of properties and methods for this class appears in Figure 10.30.

The analysis and design for the RF analyzer will be very similar to the audio analyzer. All measurement-related properties will not be cached, and the IEEE 488.2 method will not be available to the user through Get and Set methods. The properties that users will be interested in are Power Level, x, and y. Object Analysis is pretty much complete. The logic regarding the design of this component follows directly from the preceding three objects. The class diagram appears in Figure 10.31.

Now that we have a grasp of what properties and methods the objects have available, it is time to define their interactions. A short series of interaction diagrams will complete the design of the communications analyzer. First, let's consider construction of this object. The constructor will need to create the four component objects in addition to the IEEE 488.2 Instrument. Component constructors will be simple; they do not need to call other subVIs to execute. The IEEE 488.2 Instrument needs to call the constructor for GPIB Instrument. Sequentially, the communications

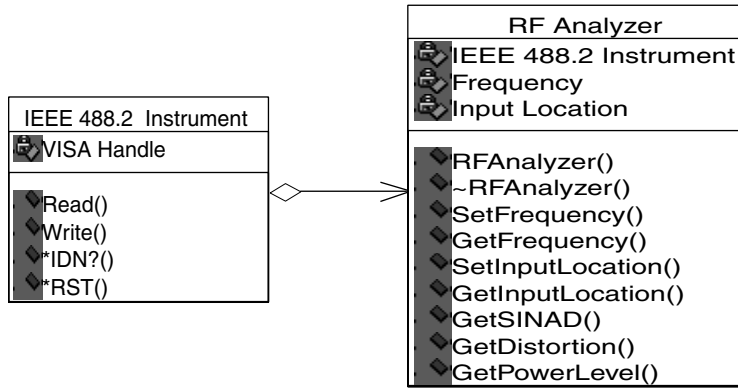


FIGURE 10.31

analyzer constructor will need to call the IEEE 488.2 constructor first; this property is handed to the component objects. The IEEE 488.2 constructor will call GPIB Instrument's constructor, and, lastly, the other four component constructors' can be called in any order. The sequence diagram for this chain of events is presented in Figure 10.32.

Figure 10.33 and 10.34 depict how the audio generator should behave when the audio generator's Set Frequency method is invoked. First, we will consider the call chain that should result when the new frequency value is equal to the old value. The method should determine that the property value should not be altered and the function should simply return. Figure 10.33 shows this interaction. Secondly, when the property has changed and we need to send a command to the physical instrument, a nontrivial sequence develops. First is the "outside" object placeholder. Set Frequency will then call IEEE 488.2's Write method to send the string. IEEE 488.2 will, in turn, call GPIB Instrument to send the string. This is a fairly straightforward call stack, but now that we have a description of the logic and sequence diagram there is absolutely no ambiguity for the programming. Figure 10.34 shows the second call sequence. Most of the call sequences for the other methods and properties follow similar procedures.

Now that each class, property, and method have been identified and specified in detail, it is time to complete this example by actually writing the code. Before we start writing the code, an e-mail should be sent to the project manager indicating that another milestone has been achieved. Writing code should be fun. Having a well-documented design will allow programmers to implement the design without having numerous design details creep up on them. Looking at the sequence diagrams, it would appear that we should start writing the constructors for the component objects first. Communication analyzer's constructor requires that these VIs be available, and we have already written and reused the IEEE 488.2 and GPIB Instrument objects. Audio generator's constructor diagram appears in Figure 10.35. The other constructors will be left to the exercises at the end of the chapter. Using skeleton VIs for these three constructors, we are ready to write the communication analyzer's constructor. The error cluster is used to force order of execution and realize the sequence diagram we presented in Figure 10.32. So far, this coding appears to be fairly easy; we are simply following the plan.

In implementing some of the functions for the analyzers and generators, we will encounter a problem. The Get and Set methods require that a particular screen be active before the instrument will accept the command. We left this detail out of the previous analysis and design to illustrate how to handle design issues in the programming phase. If you are collecting software metrics regarding defects, this would be a design defect. Coding should be stopped at this point and solutions should be proposed and considered. Not to worry, we will have two alternatives.

One possible solution is to send the command changing the instrument to the appropriate screen in advance of issuing the measurement command. This is plausible if the extra data on the GPIB bus will not slow the application down considerably. If GPIB bus throughput is not a limiting factor for application speed, this is the easiest fix to put in. Changes need to be made to the description of logic and that is about it. We can finish the coding with this solution.

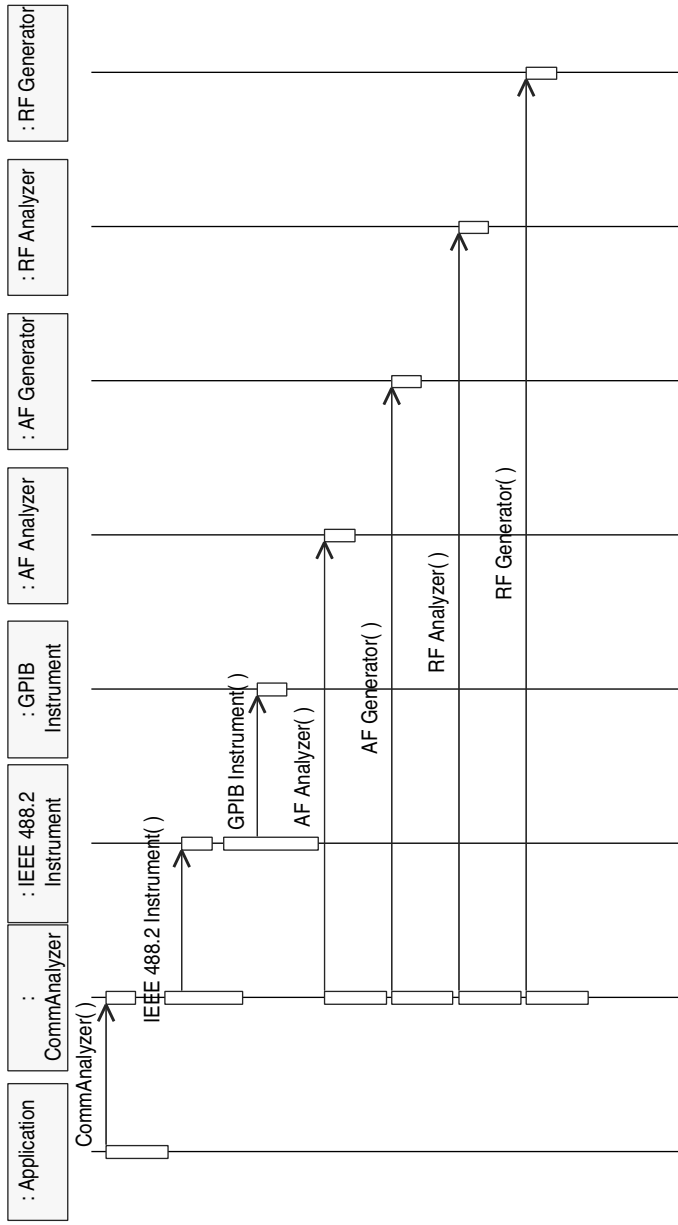

: Application
: CommAnalyzer
: IEEE 488.2 Instrument
: GPIB Instrument
: AF Analyzer
: AF Generator
: RF Analyzer
: RF Generator
CommAnalyzer( )
IEEE 488.2 Instrument( )
GPIB Instrument( )
AF Analyzer( )
AF Generator( )
RF Analyzer( )
RF Generator( )

FIGURE 10.32

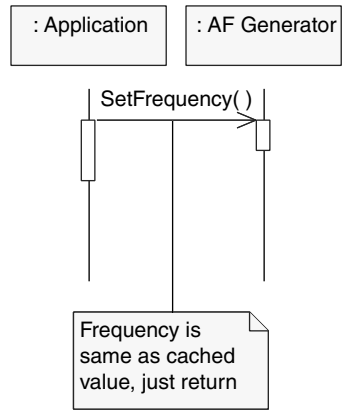


FIGURE 10.33

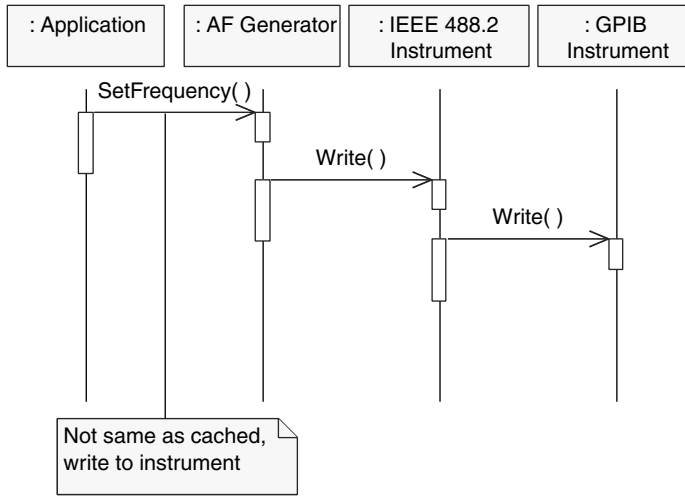


FIGURE 10.34

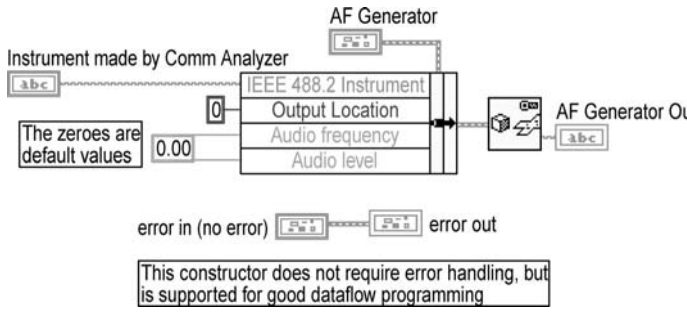


FIGURE 10.35

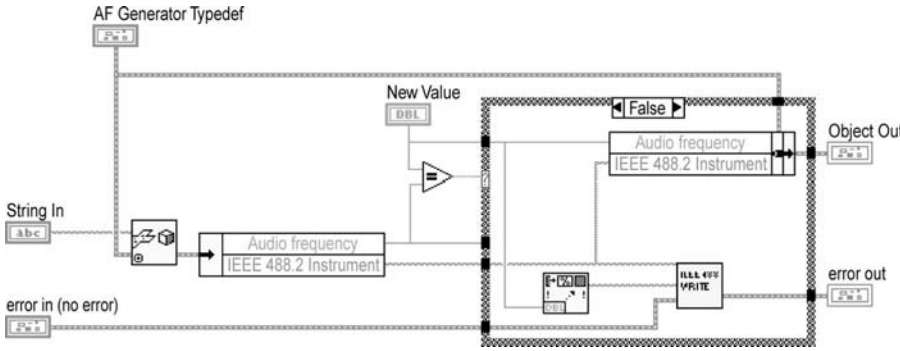


FIGURE 10.36

Another alternative is to include a screen property in the communications analyzer object. This property can cache information regarding which screen is active. If this is the case, programmers can Get/Set Screen before invoking the methods of the component objects. This will require us to go back and change all the sequence diagrams, the property list of communications analyzer, and document that users are responsible for the status of the screen.

The last alternative is to have communications analyzer define a global variable that all the component objects can access to check the status of the physical instrument's screen. This solution is going to be tossed out because it is a "flagrant technical foul" according to the object-oriented programming paradigm. Global data such as this can be accessed by anyone in any location of the program. Defensive programming is compromised and there is no way for the component objects to validate that this variable accurately reflects the state of the physical instrument. Having considered these three alternatives, we will go with the first and easiest solution. Measurement settling times will be orders of magnitude longer than the amount of time to send the screen change command. The transit time for the screen command is close to negligible for many applications. With this simple solution we present Figure 10.36, the code diagram for setting the audio generator's frequency. Remaining properties are implemented in a similar fashion.

10.8 OBJECT TEMPLATE

Many of the VIs that need to be implemented for an object design need to be custom written, but we can have a few standard templates to simplify the work we need to do. Additionally, objects we have written will begin to comprise a "trusted code base." Like other collections of VIs, such as instrument drivers, a library of VIs will allow us to gain some reduction in development time by reuse. The template VIs built in this section are available on the companion CD.

The template for constructors needs to be kept simple. As each constructor will take different inputs and use a different Typedef cluster, all we can do for the boilerplate is get the Flatten to String and Output terminals wired. Each constructor should return a string for the object and an error cluster. Simple objects such as our

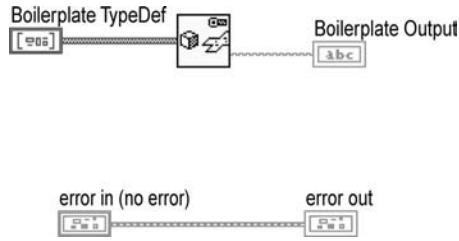


FIGURE 10.37

geometric shapes may not have any significant error information to return, but complex classes that encapsulate TCP conversations, Active X refnums, or VISA handles would want to use the error cluster information. Figure 10.37 shows the template constructor.

Get properties will be the next template we set up. We know that a string will be handed into the control. It will unflatten the string into the Typedef cluster and then access the property. This operation can be put into a VI template. It will not be executable, as we do not know what the Typedef cluster will be in advance, but a majority of the "grunt work" can be performed in advance. Because return types can be primitive or complex data types, we will build several templates for the more common types. Figure 10.38 shows the Get template for string values. This template will be useful for returning aggregated objects in addition to generic strings. Error clusters are used to allow us to validate the string reference to the object.

Set property templates will work very much like Get templates. As we do not have advance information about what the cluster type definition will be, these VIs will not be executable until we insert the correct type definition. Again, several

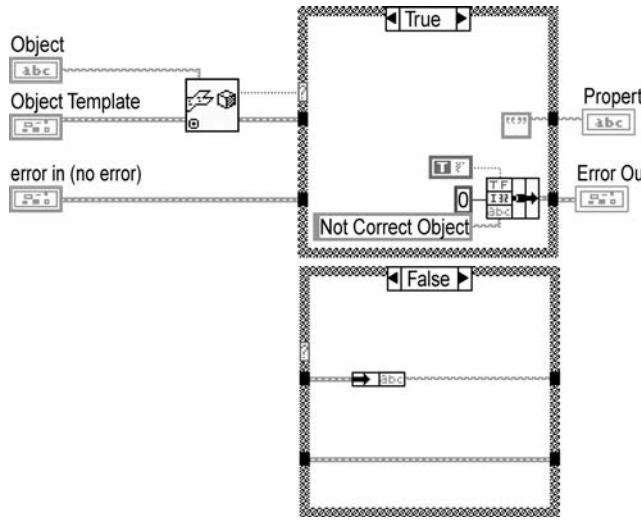


FIGURE 10.38

templates will be designed for common data types. Error clusters are used to validate that the string handed into the template is the correct flattened cluster.

10.9 EXERCISES

1. An object-based application for employee costs needs to be developed. Relevant cost items are salary, health insurance, dental insurance, and computer lease costs.
2. What properties should be included for a signal simulation object? The application will be used for a mathematical analysis of sinusoidal signals.
3. Construct an object-based instrument driver for a triple-output power supply based on the example power supply in Section 10.7.

BIBLIOGRAPHY

Bertand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ, 1998.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

LIMITED WARRANTY

Taylor & Francis Group warrants the physical disk(s) enclosed herein to be free of defects in materials and workmanship for a period of thirty days from the date of purchase. If within the warranty period Taylor & Francis Group receives written notification of defects in materials or workmanship, and such notification is determined by Taylor & Francis Group to be correct, Taylor & Francis Group will replace the defective disk(s).

The entire and exclusive liability and remedy for breach of this Limited Warranty shall be limited to replacement of defective disk(s) and shall not include or extend to any claim for or right to cover any other damages, including but not limited to, loss of profit, data, or use of the software, or special, incidental, or consequential damages or other similar claims, even if Taylor & Francis Group has been specifically advised of the possibility of such damages. In no event will the liability of Taylor & Francis Group for any damages to you or any other person ever exceed the lower suggested list price or actual price paid for the software, regardless of any form of the claim.

Taylor & Francis Group specifically disclaims all other warranties, express or implied, including but not limited to, any implied warranty of merchantability or fitness for a particular purpose. Specifically, Taylor & Francis Group makes no representation or warranty that the software is fit for any particular purpose and any implied warranty of merchantability is limited to the thirty-day duration of the Limited Warranty covering the physical disk(s) only (and not the software) and is otherwise expressly and specifically disclaimed.

Since some states do not allow the exclusion of incidental or consequential damages, or the limitation on how long an implied warranty lasts, some of the above may not apply to you.

DISCLAIMER OF WARRANTY AND LIMITS OF LIABILITY

The author(s) of this book have used their best efforts in preparing this material. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. Neither the author(s) nor the publisher make warranties of any kind, express or implied, with regard to these programs or the documentation contained in this book, including without limitation warranties of merchantability or fitness for a particular purpose. No liability is accepted in any event for any damages, including incidental or consequential damages, lost profits, costs of lost data or program material, or otherwise in connection with or arising out of the furnishing, performance, or use of the programs in this book.

Electrical Engineering

Bitter • Mohiuddin • Nawrocki

LabVIEW™ Advanced Programming Techniques

SECOND EDITION

Whether seeking deeper knowledge of LabVIEW™'s capabilities or striving to build enhanced VIs, professionals know they will find everything they need in LabVIEW: Advanced Programming Techniques. Updated to reflect the functionalities and changes made to LabVIEW Version 8.0, this second edition delves deeply into the enhancements that continue to make LabVIEW one of the most popular and widely used graphical programming environments across the engineering community.

LabVIEW's new features are by no means trivial and neither are the updates made to the new edition of this popular bestseller. The authors introduce the changes to the front panel controls, the Standard State Machine template, new drivers, the instrument I/O assistant, new error handling functions, hyperthreading, and Express VIs. A new chapter mirrors the introduction of the Shared Variables function in LabVIEW 8.0 and a new section explores the LabVIEW project view. The chapter on ActiveX was revised to include discussion of the Microsoft™ .NET® framework and new examples of programming in LabVIEW using .NET. Numerous illustrations and step-by-step explanations provide hands-on guidance.

Reflecting the latest features of LabVIEW 8.0, this second edition…

- Presents a wide-ranging compendium of advanced programming techniques incorporating the new features and enhancements of LabVIEW 8.0
- Offers self-contained chapters to let you select the coverage best suited to your needs
- Discusses LabVIEW techniques for programming within the .NET framework
- Includes a new chapter exploring the new shared variable feature introduced in LabVIEW 8.0
- Examines the new LabVIEW project view in detail

Reflecting not only changes to LabVIEW but also to the programming environment in general, **LabVIEW: Advanced Programming Techniques, Second Edition** remains an indispensable resource to help programmers take their LabVIEW knowledge to the next level.

3325

ISBN 0-8493-3325-3

90000

9 780849 333255

CRC Press
Taylor & Francis Group
an informa business
www.taylorandfrancisgroup.com

6000 Broken Sound Parkway, NW
Suite 300, Boca Raton, FL 33487
270 Madison Avenue
New York, NY 10016
2 Park Square, Milton Park
Abingdon, Oxon OX14 4RN, UK

www.crcpress.com