Bitter, Rick et al "Drivers"
*LabVIEW Advanced Programming Techniques*
Boca Raton: CRC Press LLC, 2001
This chapter discusses LabVIEW drivers. A driver is the bottom level in the three-tiered approach to software development; however, it is possibly the most important. If drivers are used and written properly, the user will benefit through readability, code reuse, and application speed.

LabVIEW drivers are designed to allow a programmer to direct an instrument, process, or control. The main purpose of a driver is to abstract the underlying low-level code. This allows someone to instruct an instrument to perform a task without having to know the actual instrument command or how the instrument communicates. The end user writing a test VI does not have to know the syntax to talk to an instrument, but only has to be able to wire the proper inputs to the instrument driver.

The following sections will discuss some of the common communication methods that LabVIEW supports for accessing instruments and controls. After the discussion of communication standards, we will go on to discuss classifications, inputs and outputs, error detection, development suggestions, and, finally, code reuse.

The standard LabVIEW driver will be discussed first. This standard driver is the basis for most current LabVIEW applications. In an effort to improve application performance and flexibility, a new style of driver has been introduced. The Interchangeable Virtual Instrument (IVI) driver is a new driver technology and will be described in depth later in this chapter.

5.1 COMMUNICATION STANDARDS

There are many ways in which communications are performed every day. Communication is a method of sharing information. People can share information with each other by talking, writing messages, sign language, etc. Just as people have many different ways to communicate with each other, software applications have many ways to communicate with outside entities. Programs can talk to each other, to instruments, or to other computers. The following communication standards are just some of the methods LabVIEW uses to communicate with the outside world.

5.1.1 GPIB

The General Purpose Interface Bus (GPIB) is a standard method of communication between a computer/controller and test equipment. The GPIB consists of 16 signal lines and 8 ground return lines. The 16 signal lines are made up of 8 data lines, 5 control lines, and 3 handshake lines. The GPIB interface was adopted as a standard (IEEE 488). The maximum GPIB data transfer rate is about 1Mbyte/sec. A later
version of the standard with added features was defined in 1987. This standard is the ANSI/IEEE 488.2. This enhancement to the standard defines how the controller should manage the bus. The new standard includes definitions of standard messages for all compliant instruments, a method for reporting errors and other status information, and the protocols used to discover and configure GPIB 488.2 instruments connected to the bus.

HS488 is a new standard that has emerged. This standard is an extension of the IEEE 488 standard and increases the GPIB data transfer rate. By using HS488 controllers and compatible instruments, the data transfer rate can be increased up to 8 Mbytes/sec. The biggest benefit of the higher data transfer rate is the use of instruments that return large data sets. Instruments such as oscilloscopes and spectrum analyzers send large amounts of data to the application computer. The HS488 standard allows you to increase your test throughput.

There are two types of GPIB commands. There are device-dependent messages and interface messages. Device-dependent messages contain programming instructions, data measurements, and device status. Interface messages execute the following operations: initializing the bus, configuring remote or local settings on the device, and addressing devices.

Many current instrument manufacturers have standardized remote commands. This allows the user of an instrument to learn how to program an instrument in a shorter period of time and makes instruments more interchangeable. In order to try to make programming instruments easier, an SCPI (Standard Commands for Programmable Instrumentation) command set was developed. The SCPI commands are for basic functions that almost all instruments support. There are a number of instruments on the market that are not SCPI compliant. These instruments have their own command sets, and formats. This can make writing automation software difficult. One example is the T-BERD PCM analyzer. This instrument is not SCPI compliant. If you wanted to reset the instrument, you would have to search through the reference manual for the command, if it exists. In this instance, to reset the instrument, you would have to write “FIRST POWER UP” to the instrument. Not only is the command not obvious, but it would require the developer to spend time hunting down commands. Figure 5.1 illustrates the GPIB driver.

FIGURE 5.1
In the Instrument I/O section of the Functions palette there are two subpalettes that contain GPIB drivers. The first subpalette (GPIB) contains the traditional GPIB 488 commands; the second subpalette (GPIB 488.2) contains GPIB 488.2 commands. The VIs from these subpalettes can be used in conjunction with a GPIB 488.2 instrument. If the instrument you are using is not GPIB 488.2 compliant, you can only use the VIs in the traditional GPIB palette.

The primary VIs in the GPIB palette are GPIB Read and GPIB Write. These two VIs are the basis for any program using GPIB instruments. There are also VIs used to wait for a service request from the instrument (Wait for GPIB RQS), obtain the status of the GPIB bus (GPIB Status), and initialize a specific GPIB bus (GPIB Initialization). Among the remaining GPIB VIs, there is a GPIB Miscellaneous VI. This VI allows you to execute a low-level GPIB command. The GPIB palette is shown in Figure 5.2.

The GPIB 488.2 palette contains additional functions. The GPIB functions are broken into five categories: single device functions, multiple device functions, low-level I/O functions, bus management functions, and general functions. The single device functions are VIs that communicate with a specific instrument or device. Some of the functions include Device Clear, Read Status, and Trigger. The multiple device functions communicate with several devices at the same time. The VIs define which devices to communicate with through an array of addresses that are input. This category of VIs includes VIs to clear a list of devices, enable remote, trigger a list of VIs, and VIs to perform serial or parallel polls of the devices.

Low-level I/O VIs allow you to have more control over communications. The VIs in this category include functions to read or write bytes from a device, send GPIB command bytes, and configure a device in preparation to receive bytes. The Bus Management functions are VIs used to either read the status of the bus or to perform functions over the entire GPIB. The VIs in this category include VIs to find all listeners on the GPIB, to reset the system, to determine the state of the SRQ line, and to wait until an SRQ is asserted. Finally, the general functions are used to make an address or to set the timeout period of the GPIB devices. The GPIB 488.2 palette is shown in Figure 5.3.

5.1.2 SERIAL COMMUNICATIONS

Serial port communications are in wide use today. One of the advantages of serial communication versus other standards like GPIB is availability: every computer has
a serial port. Another benefit to serial communications versus GPIB is the ability to control instruments at a greater distance. The serial standard allows for a longer cable length.

The most common serial standard is RS-232C. This protocol requires a transmit, receive, and ground connection. There are other lines available for handshaking functions, but they are not necessary for all applications. Macintosh serial ports use RS-422A protocols. This protocol uses an additional pair of data lines. Due to the additional data lines, the standard is capable of transmitting longer distances and faster speeds reliably. There are other serial protocols available, but those are the most widely used at this time.

The serial port VIs are in the Instrument I/O section of the Functions palette. This subpalette consists of VIs used to read data from the serial port, write data to the serial port, initialize the serial port, return the number of bytes available at the serial port, and to set a serial port break. The Serial Port Initialize VI allows you to configure the serial port’s settings. In order to have successful communications between a serial port and a device, the settings of the port should match the device settings. The settings available are buffer size, port number, baud rate, number of data bits, number of stop bits, data parity, and a flow control cluster. This flow control cluster bundles together a number of parameters, including a number of handshaking settings. The Serial VI palette is shown in Figure 5.4.

A programmer using the serial standard must ensure that the serial write does not overflow the buffer. Another issue is making sure all of the data is read from the serial port. There are a number of LabVIEW built-in functions designed to configure the buffer size and to query the number of bytes available at the serial port. Figure 5.5 shows a VI written to read information from the serial port. This VI performs the read until all of the desired data has been read.

There are additional serial port standards, which would require a separate discussion. These standards are the Universal Serial Bus (USB) and Firewire (IEEE 1394). USB allows you to plug devices into a common port, and gives you the ability to “hot swap” instruments. There are a number of hardware devices available that are USB capable. In addition, National Instruments builds devices to take advantage of this technology, including a GPIB-to-USB Controller. This external box connects to the PC through the USB port and allows the user to connect up to 14 GPIB instruments without having to have a GPIB port on the PC. This is especially useful.
when using a laptop computer without I/O slots; the controller can plug into the USB port.

Firewire allows hot-swapping of devices and can daisy-chain up to 16 devices. The main benefit to Firewire is speed. The Firewire standard boasts speeds of 100, 200, and 400 Mbits/Sec. Revisions to the IEEE 1394 standard will increase the data transfer rate to 3.2 Gbits/Sec.

5.1.3 VXI DISCUSSION

VME Extensions for Instrumentation (VXI) is a standard designed to support instrument implementation on a card. VME is a popular bus architecture capable of data rates of 40MB/s. VXI combines the speed of the VMEbus with the easy-to-use command set of a GPIB instrument. The goal of VXI instrumentation is to produce a small, cost-reduced hardware system with standardized configuration and programming. The VXI Plug&Play standards promote multivendor interchangeability by standardizing the instrument commands for all VXI instruments. By implementing instruments on cards, the size necessary to implement a test station can be greatly reduced. The ability to implement a number of instruments in a small frame allow the test developer to create a test site in places that were not practical before, freeing up resources for other applications. The VXI standard also gives the user the flexibility of custom solutions. Cards can be made and utilized to implement solutions that are not available off the shelf.

The VXI VIs are contained in a subpalette of the Instrument I/O palette. Within the VXI palette, there are 12 subpalettes. Each of these subpalettes contains specific classes of VXI drivers. The subpalettes and their contents are described in Table 5.1.
Virtual Instrument Software Architecture (VISA) is a standard Application Programming Interface (API) for instrument I/O communication. VISA is a means for talking to GPIB, VXI, or serial instruments. VISA is not LabVIEW specific, but is a standard available to many languages. When a LabVIEW instrument driver uses VISA Write, the appropriate driver for the type of communication being used is called. This allows the same API to control a number of instruments of different types. A VI written to perform a write to an instrument will not need to be changed if the user switches from a GPIB to a serial device. Only the resource name must be modified where Instrument Open is used.

Another benefit of using VISA is platform independence. Different platforms have different definitions for items, like the size of an integer variable. The programmer will not have to worry about this type of issue; VISA will perform the necessary conversions. Figure 5.6 is a side-by-side comparison of GPIB and a VISA Driver.

As is seen in Figure 5.6, the main work in a VISA application is in the initialization. GPIB communications require the address string to be passed everywhere a driver is called. If there were a change in the instrument, like using a serial instrument instead of a GPIB instrument, a large application would require consid-

<table>
<thead>
<tr>
<th>Palette Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Configuration</td>
<td>VIs in this palette are used to control the VXI library, find a device’s local address, read a device’s information table in multiple formats (16-bit unsigned, 32-bit unsigned, or string), and set a device’s information table in multiple formats.</td>
</tr>
<tr>
<td>Word Serial Commander</td>
<td>This set of VIs is used to write data to a device, read data from a device, perform command handling, or configure timeouts.</td>
</tr>
<tr>
<td>Word Serial Servant</td>
<td>These VIs deal with getting information from the device to the handler/controller. They also control the interrupt functions.</td>
</tr>
<tr>
<td>High-Level VXIbus Access</td>
<td>These VIs read or write information to a specified VXI address or register. There is also a VI used to copy a block of memory.</td>
</tr>
<tr>
<td>Low-Level VXIbus Access</td>
<td>Drivers are used to create a window into VXI address space using pointers. By using the pointers, data can be read or written. There are additional VIs for configuration settings.</td>
</tr>
<tr>
<td>Local Resource Access</td>
<td>VIs to read and set MODID lines on the backplane of the VXI bus, and read and write data from VXI local registers.</td>
</tr>
<tr>
<td>Shared Memory Access</td>
<td>VIs used to control allocation of memory from the VXI shared RAM memory area of the local CPU.</td>
</tr>
<tr>
<td>VXI Signal</td>
<td>Drivers for using VXI signals.</td>
</tr>
<tr>
<td>VXI Interrupt</td>
<td>VIs for VXI interrupt usage.</td>
</tr>
<tr>
<td>VXI Trigger</td>
<td>VXI trigger functions.</td>
</tr>
<tr>
<td>System Interrupt Handler</td>
<td>System level interrupt functions.</td>
</tr>
<tr>
<td>VXIbus Extender</td>
<td>Maps the triggers, interrupts, or utility bus signals into or out of the mainframe.</td>
</tr>
</tbody>
</table>
erable changes. All the drivers would have to be changed. An application using VISA would only require changing the input to the VISA Open VI. The resulting instrument reference would still be valid for the VISA drivers, requiring no change. VISA drivers offer flexibility.

The VISA driver VIs are located in the Instrument I/O section of the Functions palette. The VISA subpalette contains a wide range of program functions. At the top of the palette, there are a group of “Easy” VISA functions. These functions are designed for performing I/O communications with devices to evaluate functionality. These VIs are not intended for use in an application. These VIs open a session to a device that requires closing. Each time these VIs are run, a session will be opened. This can be very time-consuming; therefore, the VIs should only be used when necessary. The Easy VISA Find Resources VI can be helpful for system debugging. It will return a list of all GPIB, Serial, and VXI devices that are connected to your machine, and their resource names.

The next set of VIs are the standard VISA drivers. These VIs allow you to open a communication session, read and write data, assert a trigger, and close communications. In addition to the standard VISA VIs, there are four subpalettes with additional VIs. The first subpalette is the Interface subpalette. It contains VIs used to deal with interface-specific needs. There are VIs to set the serial buffer size, flush the serial buffer, and send a serial break. The VISA GPIB Control REN (Remote Enable) VI allows you to control the REN interface line based on the specified mode.
The VISA VXI CMD or Query VI allows you to send a command or query, or receive a response to a previously sent query based on the mode input.

The next subpalette is the Event Handling palette. The VIs in this palette act on specified events. Examples of events are triggers, VXI signals, or service requests. The High-Level Register Access subpalette allows you to read, write, and move specified-length words of data from a specified address. The Low-Level Register Access subpalette allows you to peek and poke specified bit length values from specified register addresses. The VISA palette is shown in Figure 5.7.

5.1.5 DDE

Dynamic Data Exchange (DDE) is a method of communication between Windows applications. In DDE communications, there is a server and a client application. The DDE client is the program that is requesting data or sending a command to the DDE server. Assuming both applications are open, the client first establishes communication with the server. Connections are called “conversations.” The client can then request the server to send or modify any named data. The client can also send commands or data to the server. A client can either request data or request to be advised of data changes for monitoring purposes. Like the other forms of communication, when all tasks have been completed, the client must close communication with the server.

LabVIEW can act as the server or the client. One example of LabVIEW acting as a client would be a VI that obtains data from an Excel spreadsheet or writes the data to the spreadsheet. If LabVIEW is acting as a server, another Windows program could open and run a VI, taking the data obtained to perform a task.

There are few new applications using DDE due to DDE’s limited abilities. With the development of OLE and ActiveX technologies, DDE is mainly around for backward compatibility. Keep this in mind when developing future applications using DDE.

The DDE VIs are in the Communications palette. There are VIs for opening and closing conversations, and performing advise functions, requests, and executions. In addition to the DDE function drivers, there is a subpalette contained in the DDE palette. This subpalette contains the DDE server functions. These functions are used
to register and unregister DDE service and items. There are also VIs used to set and check items. The DDE subpalette is shown in Figure 5.8.

5.1.6 OLE

Object Linking and Embedding (OLE), or automation, is the ability to place objects from other software programs into another application. This ability allows both the expansion of the program’s abilities and the ability to manipulate data in another application. An example of this would be taking a movie clip (AVI file) and embedding it in a Word file. Even though Word has no idea what a movie clip is, it can display it in the word processing environment. OLE is a method by which objects can be transferred between applications. OLE works with objects using a standard known as the Component Object Model (COM). The COM standard defines common ways to access application objects to determine if an object is in use, is error reporting, or if there is object exchange between applications, and a way to identify objects to associate them with specific applications.

OLE is a superset of the ActiveX standard and uses the same VIs. The ActiveX subpalette is within the Communications palette. The palette contains VIs to open and close automation refnums to objects. The specified object is chosen through the automation refnum input, or by right-clicking on the Automation Open VI and choosing Select ActiveX Class. In order to access an object on a remote machine, the machine name must be specified. Accessing an object on a remote machine uses the Distributed Component Model (DCOM). There is also a driver used to convert an ActiveX variant to a G data type. The Property Node function and Invoke Node function are in this palette. These functions are used to access an object’s properties and methods. In LabVIEW 5.1, there is a subpalette inside the ActiveX palette for ActiveX events. These event functions allow you to create event queues, perform event functions, and destroy event queues. There is an in-depth discussion of ActiveX with examples in Chapters 7 and 8.

5.1.7 TCP/IP

There are three main protocols for communication across networks. Transmission Control Protocol (TCP), Internet Protocol (IP), and User Datagram Protocol (UDP). TCP is built on top of IP. TCP breaks the data into the packets for the IP to send.
TCP also performs data checking to ensure the data arrives at its destination in a singular, complete form. TCP/IP data consists of 20 bytes of IP information, followed by 20 bytes of TCP information, followed by the data being sent. The TCP/IP protocol can be used on all platforms of LabVIEW and BridgeVIEW.

Every computer on an IP network has a unique Internet address. This address is a 32-bit integer, usually represented in the IP dotted-decimal notation. The address is separated into 8-bit integers separated by decimal points. The Domain Name Service (DNS) system is a database of IP addresses associated with unique names. For instance, a user looking up the National Instruments Web site (www.natinst.com) will be routed to the appropriate IP address that corresponds to the name. This process is known as “hostname resolution.”

There are a number of standards using TCP/IP that can be implemented using LabVIEW. Telnet, SMTP, and POP3 are a few applications built using the TCP/IP protocol. Telnet can be used for providing two-way communications between a local and remote host. POP3 and SMTP are used to implement mail applications.

With TCP/IP, the configuration of your computer depends on the system you are working on. With Windows 95/NT, UNIX, and Macintosh Version 7.5 and later, TCP/IP is built in. For earlier versions of Macintosh Operating systems, the MacTCP driver needs to be installed. For Windows 3.1, an Ethernet card, the drivers for the card, and the Winsock DLL must be installed. Windows 3.1 needs a third-party DLL. Win95/98/NT provide the Winsock.dll in the installation.

The TCP palette is located in the Communication section of the Function palette. The VIs in the TCP palette allow you to open and close connections. Once the connection is opened, you can read and write data through the VIs in the TCP palette. There are also VIs to create a listener reference and wait on listener. The IP to string function allows you to convert an IP address to a string. There is an input to this function to specify if the address is using dot notation. A function to convert a string to an IP address is also available. The VIs in this palette are shown in Figure 5.9.

5.1.8 DataSocket

DataSocket is a programming technology that facilitates data exchange between applications and computers. Data can easily be transferred between applications over an Internet connection. DataSocket is built using TCP/IP and ActiveX/COM technologies. The DataSocket server can reside on the local machine or on another machine on the network. You can read data using DataSocket HTTP, FTP, and local files. DataSocket can also read in live data through a DSTP (DataSocket transfer
protocol) connection. You also have the ability to control your LabVIEW application through a Web interface by using CGI functions with DataSocket.

The DataSocket VIs are in a subpalette of the Communication section of the Function palette. The DataSocket VIs work in the same way VISA or other standard LabVIEW VIs operate. There are VIs for opening and closing connections. The Open function will open communication based on the URL input and the access mode input. The URL input must be one of the above mentioned protocols. The output of the Open function is a DataSocket reference. This reference is used in the same manner as a typical connection refnum. The remaining VIs use this reference to perform actions on the desired information. You can then read or write a string, Boolean, integer, or a double value. If you want to read or write arrays of these data types, the necessary VIs are available in the DataSocket Write and the DataSocket Read subpalettes. The Advanced subpalette gives you the ability to read or write variants. In addition to the variant functions, there are also low-level functions for performing DataSocket communication. These functions include VIs to connect and update data. Finally, there is a VI to control the DataSocket server programmatically. You should also be able to access the DataSocket server from your Start menu under the National Instruments DataSocket name. The DataSocket function palette is shown in Figure 5.10.

If you want to perform live data updates, you first need to determine if the DataSocket server is running on the local machine. The typical format for a local write data to a DataSocket server is dstp://localhost/test. This assumes that “test” is the label for the data you are writing to the server. If you are using a local server, the DataSocket server will need to be launched through the function in the DataSocket Advanced subpalette. Then, you will need to open a DataSocket connection with Write Attribute selected. You can then write the data you want to share to the DataSocket server. If you are running the DataSocket server on another computer, the machine address will need to be in the DSTP address.

To read the data from the server, you will again need to determine if the server is local or on a remote machine. Once you have the server name resolved, and have a connection open to the server with the read attribute, you can use the Read DataSocket VIs to read the data in. You will need to use the Update data VI if you want to read new data after it has been written to the server.

![DataSocket function palette](image)
To read and write static data, the process is the same. The only difference is the URL used to connect to the DataSocket. Examples of a VI used to generate live data to the DataSocket server, and a VI to read the data from the DataSocket server, are shown in Figure 5.11. This example includes additional attributes. This allows items like time and date stamps to accompany the data that is being transferred. The DataSocket server is launched on the same PC as the Data Write VI. There are additional examples in the LabVIEW on-line reference.

5.1.9 DAQ

Data acquisition (DAQ), in simple terms, is the action of obtaining data from an instrument or device. In most cases, DAQ is performed using plug-in boards to collect data. These plug-in boards are made by a number of manufacturers, including National Instruments. These DAQ boards perform a variety of tasks, including analog measurements, digital measurements, and timing I/O. One convenience is the ability to obtain boards for PC, Macintosh, and Sun workstations. One of the benefits of using National Instruments boards is the availability of NI-DAQ drivers for the boards. While other manufacturers’ boards are compatible with LabVIEW, the DAQ library will most likely not be compatible with the board. Most board manufacturers do provide their own drivers for their equipment; some even have drivers written in LabVIEW. Even if the code is not written in LabVIEW, DLLs can be implemented by using the Call Library function. Code Interface Nodes (CINs) can be used to implement drivers written in C source code.
If the functionality required is available with a National Instruments board, the easiest and quickest solution is to stick with the NI board. Using the NI board will allow you to use the DAQ library. The VIs in the DAQ library are frequently updated and are completely compatible with your LabVIEW application.

The Data Acquisition subpalette is a part of the Functions palette. The Data Acquisition palette is made up of six subpalettes: the Analog Input VIs, Analog Output VIs, Digital I/O VIs, Counter VIs, Calibration and Configuration VIs, and Signal Conditioning VIs. The Data Acquisition subpalette is shown in Figure 5.12. Each of the subpalettes is comprised of a number of VIs of varying complexity and functionality. There are four levels of DAQ VIs. They are Easy VIs, Intermediate VIs, Utility VIs, and Advanced VIs. As a rule, the Utility VIs are stored in their own subpalette. The Advanced DAQ VIs are also stored in their own subpalette. The main difference between the Easy VIs and the Intermediate VIs is the ability of the Easy VIs to run as stand-alone functions. These VIs call the higher-level VIs to perform the task. The Easy VIs allow you to pass in the device number and channel numbers. The VIs will also perform error-handling functions to alert you if an error has been encountered.

The Analog Input subpalette is shown in Figure 5.13. The palette consists of the four types of VIs described above. The Easy VIs include functions to acquire one or multiple waveforms from an analog input. There are also functions for acquiring samples at the designated channels. The Intermediate VIs allow you to configure the hardware and associated settings, start an acquisition, read the buffered data, make single scan acquisitions, and clear the analog input task. The Analog Input palette contains two subpalettes. The first subpalette contains the Utility VIs. These VIs include functions to initiate a single scan, a waveform scan, or a continuous scan. The second palette contains the Advanced functions. The Advanced function palette contains VIs to perform configurations, read the buffer, set parameters, and control analog input tasks. We could devote a number of chapters on DAQ functions, but the DAQ functions are described in great detail in the *Data Acquisition Basics* manual. We will not attempt to cover material that is concisely covered already.

5.1.10 File I/O

File input and output is a type of driver that people do not often think of. The ability to read data from a file and write data to a file in many ways is similar to reading
data from and writing data to a GPIB instrument. You require a means to identify the file you want to communicate with. Instead of a GPIB address you have a file path. You also need to be able to transfer data from one place to another. Instead of passing data between the computer and the GPIB instrument, you are passing data between the LabVIEW program and a file. The File I/O functions are very similar to instrument or communication drivers.

The File I/O VIs can be found in the File I/O section of the Function palette. This subpalette contains a number of file functions as well as subpalettes containing VIs pertaining to binary files, file constants, configuration files, and advanced file functions. The standard file I/O functions include VIs for opening/creating a file, reading data from a file, writing data to a file, and closing a file. In addition to these functions, there are VIs for writing and reading data from a spreadsheet file, writing or reading characters from a file, and reading lines from a file. The File I/O palette is shown in Figure 5.14.

There are two remaining functions that are included with the standard file I/O functions. The first VI allows you to build a file path. This VI creates a new file path by appending the file name or relative path from the string input to the base path. The default value of the base path is an empty path. The result is the combined file path. If there is a problem in one or both of the inputs, the VI will return “not-a-path.” The second function takes a file path and breaks it apart. The last section of the path is wired out as a string filename. The remainder of the path is wired out as a path. The VI will output an empty string and “not-a-path” if there is an invalid input. The binary file VIs allow you to read and write 1- or 2-D arrays of data to a byte stream file. The byte stream file can be in a signed word format or a single precision format. The configuration file palette contains VIs used to read and modify information in the configuration files. The File Constants palette contains VIs that allow you to access the current directories, paths, or VI library directories. In addition to these functions, there are constants that can be used to create inputs to the file I/O VIs.

The Advanced palette contains VIs that perform a number of file-related tasks. The Advanced palette is shown in Figure 5.15. The File Dialog function displays the file dialog box for the user to select a file. The output is the path of the file selected. The Open File VI allows you to specify a datalog type. There is a function used to find the offset of the end of file (EOF). The seek function allows you to begin a file in a position other than the beginning of the file. There are VIs used to
set access rights for a specified file, as well as to find out information on the file, directory, or volume.

There is a set of five VIs in the Advanced palette that performs actions on directories. There is a VI that allows you to move a file or directory. There are also VIs that allow you to copy a file or directory, as well as delete a file or directory. The New Directory function allows you to create a directory at the specified path. The List Directory function lists all of the file names and directory names that are found in the directory path.

The final set of functions in the Advanced palette are VIs used to convert between strings and paths. The functions can perform the functions on a single string or an array of strings. There is also a VI that converts a refnum to a path. These VIs are useful when converting string paths created by the user in a user interface to a file path to perform file functions.

We will now give a quick example of how to read and write data when dealing with datalog files. The first step is to create the data type used for storing the data. For this example we will be recording three distinct values per datalog value. The first is the index of the data. This is simply the value of the For loop index used to create the data. The second item in the data cluster is the data. The data for this example is simply random numbers generated between 0 and 10. The final data type used for the cluster is a date and time stamp. This value is written as a string. To summarize, our data type consists of an integer, a real number, and a string.

The first step is to create the code to perform the data generation. The For loop executes 100 iterations. Inside the For loop, the loop index, the test data, and the time and date string are bundled into a cluster. This cluster is wired to the output of the For loop, where auto indexing is enabled. When all the data has been collected, the
New File VI is used. The File Path contains the name and location of the file you are writing the data to and will be needed when you want to retrieve the data. The file path is the only required input. There are a number of other inputs to the VI that can be wired, or left as default. To write and read datalog files, you will need to wire a copy of the data format to the datalog type. Wiring the actual data to the input, or wiring a constant with the same data type, can do this. The other inputs are permissions, group, deny mode, and overwrite. The overwrite input for our example will be given a “true” value. This allows the program to overwrite an existing file with the same name as specified in the file path input. If the input were “false,” the program would error out when trying to create a new file that already exists.

Once the file is created, the next step is to write the data out. The Write File VI is used to send the collected data to the datalog file. The inputs of the Write File VI include convert eol (end of line), header, refnum, positive mode, positive offset, error in, and the data. The only required inputs are the refnum and data inputs. The data from the For loop is wired to the data input. The final step of this subVI is to close the file using the Close File VI.

The next step is to create a VI to read the data back from the file. In this VI, the Open File function is used to create a connection to the file. The File Path input is used to point the VI to the datalog file. In addition to the file path, the data type is wired to the Datalog Type input. This data type needs to match the data type of the cluster we wrote to the file. This allows you to read the information back in the appropriate format. In addition to the datalog type and file path, you can set the open mode and deny mode for the file. This allows you to determine the file permissions. Once the file is opened, you need to use the Read File function. This VI is used to acquire the data from the file, and write the data to an indicator. Again, the final step is to close the file. The code diagram for the Datalog Write VI and the Datalog Read VI is shown in Figure 5.16.

### 5.1.11 Code Interface Node and Call Library Function

LabVIEW has the ability to execute code written in C as well as to execute functions saved in a DLL. There are two methods for calling outside code. The programmer can call code written in a text-based language like C using a Code Interface Node (CIN). The programmer also has the ability to call a function in a DLL or shared library through the use of the Call Library function. A short description of each will follow.

The CIN is similar in some respects to a subVI. The CIN is an object on the block diagram of a VI. The programmer can enter inputs required to execute a function, and wire the outputs of the CIN to the remainder of the program. The main difference is a subVI is code written in the G language to perform a function, while the CIN executes text-based code to perform the function. The CIN is linked to compiled source code. When the execution of a block diagram comes to the CIN, LabVIEW calls the executable code, returning the final outputs to the VI.

There are a number of reasons for using the Code Interface Node. One benefit is the ability to use existing code in your LabVIEW program. If a function is already written in C, you have the ability to integrate the code into your LabVIEW program
to reduce development time. Another benefit to using a CIN is to expand the functionality of LabVIEW. Certain system functions that do not have corresponding LabVIEW functions can be implemented using code written in C. This can help a programmer to perform low-level programming with LabVIEW’s graphic-based interface. A final consideration for using CINs is speed. While LabVIEW is fast enough for most programming tasks, certain time-critical operations such as data acquisition and manipulation can be done more efficiently in a programming language like C. The use of the CIN allows the programmer to use the right tool for the right job.

The ability to use prewritten code is a key to reducing development time. Functions to perform many Windows functions have already been written. These functions are typically written in C, and are stored in Dynamic Link Libraries (DLLs). LabVIEW can call these Windows functions in two ways. The first way is through the use of a Code Interface Node. An easier method for calling DLL functions is through the use of the Call Library function. The main difference between calling C code in a CIN and using the Call Library function to call a DLL is the integration of the source code. When using a DLL, the code remains in its library; it is not copied into the executable files of the application. The other obvious difference is the fact that DLLs are Windows-specific, while the Code Interface Node can be used across platforms.

For more information on the Code Interface Node, the Code Interface Reference Manual can be found on National Instruments’ Web site. The PDF file covers how to integrate a CIN on any platform. For information on using DLLs, there is an application note on the NI Web page. Application Note 087, “Writing Win32
Dynamic Link Libraries (DLLs) and calling them from LabVIEW,” discusses the methods for using DLLs.

5.2 DRIVER CLASSIFICATIONS

There are three main functions a driver performs. The three types correspond to the three main purposes of a driver: configure an instrument, take a measurement, or check the status. These three main types of drivers will be discussed below. When creating driver VIs, National Instruments recommends a standard format the drivers should follow. Driver libraries should contain the following functions: Initialize, Configure, Action/Status, Data, Utility, and Close.

5.2.1 CONFIGURATION DRIVERS

The first type of driver is a Configure VI. These VIs should open or close communications with the instrument, initialize the instrument, or configure the instrument for the desired use. The Initialize driver first performs the initial communications. This should include opening a VISA session if VISA is being used. The Initialize driver can also perform instrument setup and initial configurations. This can allow the instrument to begin in a known or standard state. The Configuration Instrument drivers send the necessary commands to the instrument to place the instrument into the state required to make the desired measurements. There may be a number of configuration VIs for a particular instrument, logically grouped by function or related purpose. The Close driver closes the instrument communication, the VISA handle, and any other required items to complete the testing process. It is important to close the instrument communications, especially when doing serial and TCP communications. When a serial port is open, no other applications can use the port. If the port is not closed, the port is inaccessible until LabVIEW is closed. With TCP, when you connect to another machine, the port on that machine will stay open unless you close the session or the session time out.

5.2.2 MEASUREMENT DRIVERS

Measurement drivers are used to take measurements or read specific data from the instrument. The user should be aware that a data driver does not always require reading data from an instrument. The data driver could also be used to provide data to an instrument, like sending a waveform to a signal generator. It is important to note that only one measurement should be taken per driver. This is done to promote reusability as well as to ensure the application speed is not compromised by taking unneeded measurements.

5.2.3 STATUS DRIVERS

The action/status drivers are used to start or stop a specified process, check errors, and general instrument-related information. One example would be a VI written to start and stop a Bit Error Rate (BER) test or a waveform capture from a spectrum

©2001 CRC Press LLC
Another example is checking a status register to find out if a test that has been initiated is completed so the result can be read from the instrument. The VI would not change any of the instrument configurations, only the initiation or termination tasks are performed. Since checking the status of an instrument can require the instrument to be reset, a set of utility drivers should also be designed. The utility drivers are used to perform tasks such as reset, self-test, etc.

5.3 Inputs/Outputs

An important aspect of a driver is the interface with the calling VIs. There are a number of standard inputs and outputs for drivers. The Error In and Error Out clusters are the most important I/Os in a driver. These clusters have three components. For the Error In cluster, the first control is a status Boolean control; a “true” indicates there is an error. The second is a numeric control to display an error code. The final control is a source string. This string can indicate where an error occurred. There are two primary reasons for using the Error In and Out clusters. The first reason is obviously error handling. If an error has already occurred in a program, the Error In cluster will pass this information to the driver, preventing the execution of the intended task. The error cluster can also pass error information out of the driver if an error occurred while the driver was executing. A discussion of error handling is described in the following section.

The second reason for using the Error In and Out clusters is flow control. The wiring of the Error Out of one VI to the Error In of another forces the order of execution because of data dependency. For example, an instrument needs to be configured prior to taking a measurement. Wiring the Error Out of the configuration driver to the Error In of the Measurement driver forces the order of execution.

The other required inputs are the instrument communication handles. Depending on the communication VIs being used, a number of different inputs could be used. We suggest using VISA standards in your drivers. This will allow the same driver format regardless of what type of communication is used to address your instrument or device. The standard method for wiring the connector pane has the VISA session in and out in the top left and right positions, respectively. The Error In and Out are in the bottom left and right positions, respectively. This consistency of location makes connections easier to wire and find.

For readability and ease of use, the programmer should use as few inputs and outputs to a driver VI as possible. The use of clusters should be avoided unless the information is packaged in a form that other subVIs would use like the error cluster. If the cluster is not passed on, the main program will need to bundle and unbundle the items. This can obscure the intention of the code and complicate the code diagram. Additionally, the complex data type will have an effect on performance.

5.4 Error Handling

Error handling is one of the most important considerations when a programming task is begun. For this reason there is an entire chapter in this book dedicated to
error handling. This section will just highlight some of the driver-specific error-handling issues.

The main error handling that should be performed in the driver is the detection of errors that are passed in. If an error is passed into a driver, the driver should not execute any tasks. The driver should consist of a case statement controlled by the status field of the error cluster. The driver code would then only execute if no error passed in. When an error is passed into a driver, the instrument communication VIs will not execute if an error cluster is passed to them. Error processing should only occur in the upper levels of the program, as prescribed by the three-tiered design architecture. The benefit of not processing errors in the driver is the ability of the driver to be reused. If error processing is performed in the driver, the results of the processing may not be applicable to a new program using this driver. Doing error processing in the driver would cut down on code reuse. An example of the use of this “bypass” is shown in Figure 5.17.

The next issue with error processing in drivers is the implementation of timeouts. A driver should have a way out. If a driver is written to read the status of a register through the use of a While loop to read the data from the device, there should be a way to exit after a specified time if the desired response does not occur. This can result in setting an error if the program will not function without the desired value.

In writing applications that read data from a device, you should add code to ensure any errors that occur during the data acquisition are handled in an appropriate manner. For example, say you are reading data from a serial instrument. In this example you are reading the information from the serial port until the desired data is read. To perform this task, the read operation is in a While loop that is executing until the desired input is read. When the desired input is received, a “false” Boolean is wired to the conditional terminal of the While loop. If an error would occur, the desired input would never be received, resulting in the While loop continuing to execute until you stop the application. You should check the Boolean value of the error cluster in each iteration of the While loop to check for an error. The result of this error check can be combined with the result of the data check to determine whether to execute another iteration of the While Loop. The Boolean from the error cluster and the data check can be combined through the Boolean logic functions to control the conditional terminal of the While loop. An example showing all three of the above-mentioned techniques is shown in Figure 5.18.

One type of error detection that should be mentioned is the ability to set error traps in the driver code for debugging purposes. During the development stages of a driver, traps can be put in place to trap and isolate errors. This can lead to faster error detection for the purpose of debugging the driver being developed. These “traps” can be either disabled or removed when the driver development has been completed. Some instances of error traps can be simply collecting the data being read in from a serial port, and saving the data to be reviewed by the developer. Since some errors will only occur when running at full speed, recording the data for later analysis could be of great benefit. The recording of this same data would be considered unnecessary in the final driver version, hence the need for an error trap. Once the drive has been fully debugged, the trap can be eliminated. Data logging,
discussed in the error-handling chapter, is a similar tool that allows you to save and view data after the VI has been executed.

When measurements are being made in a While Loop, or setup is being performed in a state machine, care needs to be taken with error handling. There should always be a shift register passing the error cluster to each iteration. When this is forgotten, errors become difficult to track because the error cluster gets cleared with the next iteration of the While or For loop.

5.5 NI SPY

It is difficult at times to debug drivers. Commands are sent to the instrument by the program, but are the parameters correct, how long do the calls take, is there a problem with the instrument, etc.? The developer performing the application debugging needs a way to monitor and verify that the program is doing what was intended. One tool provided by National Instruments can aid in code verification. The NI Spy utility is an application that monitors, records, and displays API calls made by National Instruments applications. The NI Spy can be used to locate and analyze any erroneous API calls that your application makes, and to verify that the instrument communication is correct.
5.5.1 **NI Spy Introduction**

The NI Spy program is similar to a GPIB analyzer. The NI Spy displays function call names, parameters, and GPIB status as the developers program executes calls. The NI Spy allows access to information like the contents of data buffers, process and thread IDs, and time stamps for the start and finish times of the function calls. The spy program can also create a log of the information, although this can produce a significant performance loss.

NI Spy requires a special version of the instrument drivers to work properly. The Spy-enabled versions of the National Instruments drivers are loaded when Start Capture is selected. When finished using NI Spy, you should restore the non-Spy-enabled version of the drivers. This is because the Spy-enabled drivers can slow down the performance of other applications. You should use NI Spy only while you are debugging your application or when performance is not critical. To switch back to the non-Spy-enabled drivers automatically you can select Restore Software on Exit from the Spy menu before you exit NI Spy.

5.5.2 **Configuring NI Spy**

The first step is to open the NI Spy program. If you go to the Start menu of your computer and then to the Programs folder, there should be a folder labeled “VXIpnp” and there should be an icon for the NI Spy. When this icon is selected, the window shown in Figure 5.19 comes up. In the title bar, the name “NI Spy” should appear, followed by the program’s status. In parentheses, the title bar will indicate whether capture is on or off. By default, Capture is off when you open the NI Spy application. Figure 5.19 shows the NI Spy window with Capture on.

Before starting the NI Spy program, the first step should be to configure the options for the application. By selecting the Spy menu, the following options are available to you: Start Capture, Options, Restore Software on Exit, and a list of the available API types to capture.

To modify the NI Spy capture options, select Options from the Spy menu. The NI Spy options can only be modified when Capture is off. NI Spy, by default, displays 100 calls in the Capture window, displays buffers in Brief Buffer mode, and does not enable file logging. The Call History Depth option identifies how many API calls the NI Spy will display. If more than the selected number of API calls are made, the Capture window will show the most recent calls, discarding the calls at the beginning. If the NI Spy program is unable to display all of the API calls due to low system memory, a message box will appear giving the user the option to stop the capture or free up system resources before continuing.

The Data Buffer Mode selection allows you to choose between Brief or Full Buffer mode. The Brief Buffer mode displays up to 64 bytes of data, while the Full Buffer mode displays up to 64K bytes of data. For either of these modes, if there is more data than the allowed buffer, the middle data will be removed. For example, in the Full Buffer mode, the first 32K bytes and the last 32K bytes of data will be displayed. A row of dashes between the two halves of the buffer is inserted to indicate that part of the data has been omitted.
The File Logging selection in the NI Spy options allows the program to record all calls to a log file. File logging is useful when debugging an application that causes the system to crash. If file logging is used in the Fail-Safe Logging mode, you can view the API calls that were captured prior to the system crash by opening the saved log file. In order to use this function, a file name must be provided to store the logged API calls. There are two modes of file logging available. The first is Fail-Safe Logging. Fail-Safe Logging is a method of guaranteeing that the log file will not be corrupted if the system crashes. The logging is accomplished by opening the log file, writing the data, and closing the log file after each API call. It should be obvious that this method of logging the data is slow. If performance and time are an issue, Fast Logging is available. This method of logging opens the file at the start. The data from each call is written to the log file when the call is captured. The file is not closed until the capture is stopped or logging is disabled. The Fast Logging method of file logging is much faster than Fail-Safe Logging, but if your system crashes, data will be lost.

If you have more than one National Instruments driver installed on your computer, you can specify which APIs you want to spy on at any time. The API choices are listed in the Spy menu below the Restore Software on Exit option. Types of National Instruments Drivers are GPIB-488.2, VISA, and IVI-type drivers. By default, all installed APIs are enabled. There will be a check next to the API types selected for capture. You can omit any on the list by clicking on the name; the check will be removed.
5.5.3 Running NI Spy

There are three ways to start capturing API calls. The first is to select Start Capture from the Spy menu. The second method is to click on the arrow button on the toolbar. Finally, the user can push F8 to turn Capture on. Once you turn Capture on, you can run your application. When you want to view the captured information you can return to NI Spy to view the captured calls. To turn Capture off, click on the red “X” button on the toolbar.

You can view the API calls in the main NI Spy window as NI Spy captures them. The captured API calls are displayed in the order in which they are received. There is one line of information displayed for each captured call. The information includes the number of the call, a C-style function prototype, and the start time for the call.

By using the Properties dialog box you can see detailed call information for every captured API call. To see the properties of a specific call, double-click on the call in the Capture window, right-click on the call and select properties, or select Properties from the View menu. The Properties dialog box includes one to five pages of detailed information on the captured call. All API captured calls have a General tab, most captured calls have Input and Output tabs, some captured calls have a buffer page, and some IVI captures can have an Interchange Warning tab. The General section displays the process and threads IDs, the Windows handles, and the start and stop time statistics. The Input page displays the API call's input parameter types and values. The Output section displays the parameters that were returned after the call completion. The buffer page is only present for calls that involve the transfer of a buffer of data; this page displays the contents of the data buffer. Finally, the Interchange Warning section displays warnings about the specific call with respect to instrument interchangeability. This option is available for IVI drivers.

To search through the list of captured calls to find a specific string in the API function names, parameter values, or any other string, select Find from the Edit menu. Enter the text that you want to search for in the Find What box. Press the Find Next button to find the next captured call containing the specified string. The Match Errors Only selection can be used to limit the search to captured calls that have an error. If no search string is specified, the search locates the next captured call that failed. The Match Case selection specifies whether the search is case sensitive.

5.6 Driver Guidelines

Aside from the general driver information, there are a number of implementations that can add robustness and reusability to a driver. This section will give an overview of some of the functionality that should be added to a driver to accomplish the desired results.

One guideline that should be followed is the method of only making one measurement per driver. Since the programmer will want different measurements at different times, the programmer should keep one measurement to a driver. This allows the code to be reused easily. The user of the driver will not have to take a
number of measurements in order to receive one desired value. Making multiple measurements when only one measurement is desired limits performance.

When developing a driver, the programmer should try to combine configuration settings into logical groups. If configuring an RF generator requires setting four different parameters every time, the configuration of those parameters should be in a common driver. This would allow the user to set the generator with the appropriate settings through the access of one driver.

When you are linking the controls and indicators to the connector panel of the icon, you should choose a connector configuration that will provide extra connectors. When all of the inputs and outputs have been wired, extra connectors allow for expansion without disconnecting all existing connections. When a driver is already called in a program, and if the programmer adds a new input or output, the user will not have to rewire all of the existing connections. When there are extra connectors, the existing connections do not change, allowing the current wiring to remain unchanged.

5.7 REUSE AND DEVELOPMENT REDUCTION

The biggest benefit of developing quality drivers is the ability to reuse the drivers. Even when the programmer does not expect to use a specific driver again in the future, things change quickly. There is no better feeling in software development than, when developing an application, you realize that the underlying code has already been written. If a driver has been properly written, applications that are completely different could still use the same driver. The ability to reuse code is the biggest factor in cycle-time reduction. By not having to rewrite drivers, which includes time to learn the equipment, coding, and debugging, the user can dramatically reduce the time required to develop an application. Making drivers generic enough to reuse can require more time and effort up front, but the benefits that can be realized are substantial.

There are many drivers for numerous instruments and manufacturers that have already been written. The first place you can look for an instrument driver is on the installation CD that came with your LabVIEW application. The second disk is a disk of instrument drivers. In addition to these drivers, many of the drivers are available on the National Instruments Web page. Not only is this resource a comprehensive list of drivers, but they are the most recent versions. The National Instruments ftp site is ftp.natinst.com. Your login is “anonymous” and your password is your Internet address.

Many drivers available on the National Instruments Web page have been submitted to NI and accepted for distribution. There are standards that NI requires all drivers submitted adhere to. Many of the standards have already been discussed, and these standards can be found in the application note, AN106. Since the drivers have already been designed to the required standards, they should be easily inserted into your application with no modification. This allows the programmer to concentrate on developing the application without concern about the underlying communications. This can lead to significant development time reduction.
For unusual or difficult-to-find instrument drivers, there is another resource available. The LabVIEW Info Group is a place you can try. The Info Group is a large knowledge base that you can utilize. For subscription requests you can send an e-mail to info-labview-request@pica.army.mil. To post a message to the Info Group, send an e-mail to info-labview@pica.army.mil.

5.8 DRIVER EXAMPLE

To tie together some of the driver techniques and guidelines, we will present an example set of drivers. This set of drivers will communicate with Microsoft Word using ActiveX. This example will only create a couple of relevant drivers for illustration purposes. If you want more information on ActiveX, Chapters 7 and 8 will give a detailed description and numerous examples.

The first step is to define the task we want to accomplish. We will want to open Word, create a new file, set the margins, set the page size, set the page orientation, write text to the file, save the file, and close Word. The first step is to identify the driver types needed. You will need configuration drivers and measurement drivers. Since configuration drivers perform instrument communication and configuration, the VIs needed to open Word, close Word, and configure the settings will be contained in these drivers. The action of reading or writing data to an instrument or application requires measurement VIs. The write text to file will fall into this classification.

A driver to open an automation reference to Word will need to be created. This action will be combined with the creation of a new file. This allows the user to open Word with a new document in the initial step. The next driver to be created will configure the page setup parameters. Most times when you are modifying a one-page setup parameter, you will want to modify additional page setup parameters. This is a good place to combine the configuration settings into one subVI to facilitate ease of programming. Not only will the programmer be able to see all of the input parameters that can be changed in one location, but the driver can ensure order of execution. Some of the page setup parameters need to be modified after other parameters have been set. For example, you need to modify the page style prior to setting the orientation. The orientation setting will be reset after modifying the page style. If you are placing individual VIs to set these parameters, you could forget or be unaware of certain data dependencies, causing parameters to not be set in the desired manor. The code diagram for the Page Setup Configuration VI is shown in Figure 5.20. In addition to the data dependencies there are issues with data conversions. For example, when writing a value to a margin input, you would attempt to write data in inches. However, to get a margin value of one inch, a 72 needs to be wired to the input of the property node. Inside the driver, there is a function to convert an inch input to the required automation input. This allows you to abstract this information from the person using the driver.

The Write Text VI takes a string input and inserts it into the file at the specified index. If making multiple write statements, you could wire the end value from the previous write to the start value of the current Write VI. This allows you to do incremental data storage in the file. You would only want to have this VI write the text to the file. Any additional functions added to this VI would limit your ability
to reuse the VI. For example, if you wanted to perform a spell check on the document, you would have to perform this spell check each time text is written to the file. You may only want to check the spelling after all of the text has been written to the file. If the spell check function is in its own VI, you can invoke this function when you need it. There is also the possibility you do not want to perform a spell check at all. Measurement VIs should be in their own VIs unless you are sure you will always want to do the multiple tasks together. An example using these VIs is shown in Figure 5.21. In the example, Word is opened; a new file is created (testfile); some of the page setup parameters are modified; two strings are written to the file, separated by a time delay; and the file is closed. More information on controlling Microsoft Word using ActiveX is included in Chapter 8.

5.9 IVI DRIVERS

IVI drivers were developed to allow hardware-independent test programs. In 1997, a number of manufacturing companies approached National Instruments to develop generic drivers that would be interchangeable. The IVI Foundation was a direct result of this effort. The organization, made up of representatives from National Instruments and a number of the instrument manufacturing companies including Hewlett Packard, Tektronix, Rohde & Schwarz, and Anritsu, has developed a set of
standards and requirements for “generic” drivers. The IVI Foundation is an evolving group that is open to end users and interested parties. Anyone who is interested in joining can find more information on the IVI Foundation Web site (www.ivifoundation.org).

The goal of the IVI Foundation was to build upon the standards set by the VXI Plug&Play Systems Alliance. The VXI Plug&Play standards promote multivendor interoperability by standardizing the instrument commands for all VXI instruments. IVI instruments will go one step further by trying to standardize an instrument type regardless of format. A power supply would have the same API regardless of the standard (GPIB, Serial, VXI, other) or the manufacturer.

IVI drivers are not language specific. By using DLLs to convert the commands from a uniform API to the required instrument code, there is a wide range of programming languages that can be used. LabVIEW and LabWindows/CVI are both capable of using IVI drivers; however, the DLLs can only be written using LabWindows. Due to the use of DLLs, IVI drivers are not platform independent. If you do not want to write your own drivers, or are not using LabWindows/CVI, a library of IVI drivers is available from National Instruments. In order to install the IVI driver library your PC needs to be running LabVIEW 5.1.

5.9.1 Five Classes of IVI Drivers

The initial rollout of the IVI standards encompasses five classes of IVI drivers. The five classes are the Oscilloscope, Digital Multi-Meter (DMM), Arbitrary Waveform/Function Generator, Switch, and Power Supply. New classes may be defined as the technology advances.

Let’s look at the DMM class as an example. The IVI driver for the DMM class (IviDmm) is designed to operate a typical DMM, as well as support advanced functions found in the more complex instruments. The IVI class description divides the DMM into two sections: fundamental capabilities and extensions. The fundamental capabilities cover functions like reading a measurement or setting a range. An extended capability would be like setting auto-range, making multiple measurements, or other advanced features not available on all DMMs. For the DMM, there are three groups defined. Groups refer to the defined classification of commands. Table 5.2 shows the groups defined for a DMM in the IVI documentation.

<table>
<thead>
<tr>
<th>Group Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IviDmm</td>
<td>Fundamental Capabilities: complies with the IviDmm fundamental capabilities</td>
</tr>
<tr>
<td>IviDmmMultiPoint</td>
<td>Extension Group: IviDmm with the capability to accept multiple triggers and acquire multiple samples per trigger.</td>
</tr>
<tr>
<td>IviDmmDeviceInfo</td>
<td>Extension Group: IviDmm with the capability to return attributes that give extra information concerning the instrument’s state, such as accuracy and aperture time.</td>
</tr>
</tbody>
</table>
5.9.2 INTERCHANGEABILITY

This section will discuss how IVI drivers allow for instrument interchangeability. One of the problems that has been seen in production testing for a long time is the lack of instrument interchangeability. This problem can arise for a number of reasons. An instrument that needs to be taken out to perform calibration or maintenance is one example. Other possible scenarios are when an instrument needs to be replaced and is no longer available; if the test system developer wants to use an instrument from another manufacturer; if the test software is going to be used by a group in another area with their own set of instruments. These issues are problems because the test software would have to be altered to replace an instrument with one from another manufacturer, or a newer model with new functions and commands. These problems force test system developers to stay with the same system instead of improving or cost reducing. The ability to change instruments would allow greater flexibility. The first benefit of IVI drivers is the ability to interchange instruments. A power supply from a different manufacturer can replace the existing power supply without changing the test software. This will allow the development of a generic test station; users would be able to change instruments based on availability and cost. Figure 5.22 displays the IVI hierarchy for the program, class driver, and IVI-specific driver.

5.9.3 SIMULATION

This section will discuss how an IVI driver can be used in simulation mode to allow debugging and input checking without the instrument being connected to the computer. When a programmer is developing software, the ability to incrementally debug the code is a technique that helps reduce development time. This would be an implementation of the spiral software development model. There is a full discussion of software development models (spiral and waterfall) in Chapter 4. By using IVI drivers in simulation mode, the test code can be debugged without the instrument being connected to the computer. The driver will return an instrument handle to...
allow a program using VISA to run without the instrument physically present. The user can also use the driver in simulation mode to choose the measurement that will be returned to the test program. This will allow the designer to test the program’s response to common and unusual measurements returned by the instrument. The measurement returned can be set to random number generation within a range.

When using instrument-specific drivers, another feature is realized. The developer can perform range and status checking while developing the software. The driver will verify that the inputs sent to the instrument are within the specifications of the instrument. These are options that can be turned on or off. Turning on the range-checking feature helps the developer debug the test software. Turning off range-checking allows for faster execution time when the program is run in the final environment.

5.9.4 State Management

This section discusses how an IVI driver can speed up an application when state caching is used. One problem encountered when programming a test application, particularly when utilizing state machine architecture, is the lack of knowledge of the instrument’s current state. The user will not know what state the instrument is in at a given time, requiring the programmer to set all necessary configurations, even if the instrument is already configured properly. This can add substantial time to a test application.

The solution is to use state caching. This can be performed when using LabVIEW or LabWindows/CVI Version 5.0. When using state caching, the last setting for each function on an instrument is stored. When the driver goes to change the setting of a function, the driver checks to see what the last known state of that function was. If the setting is the same, the driver will not execute the command. The driver also tracks changes in settings when different screens are displayed.

5.9.5 IVI Driver Installation

When the IVI driver CD is inserted into the drive, the IVI Driver Library Installation interface starts. In the interface you have the options of viewing the release notes, installing the IVI driver library, installing instrument drivers, and browsing the CD. To install the IVI software you will need to click the IVI Driver Library Installation selection. This will begin the standard installation interface. After making the typical selections, a selection screen will appear. The installer will prompt you to select the instrument drivers to install. This is the initial place to obtain and install the IVI instrument drivers. There are a number of items on this installer screen. On the left of the screen is a selection for the IVI class. On the right side is a listbox containing the specific drivers. In order to install the drivers you need to use in your development, you must first select the desired IVI class. This will list the available IVI drivers in the specific driver input. In the specific driver input is the list of available drivers with a checkbox selection on the left of the individual drivers. To select the needed driver, you need to select the appropriate checkbox.
In addition to the IVI class input and the specific drivers input, there are three additional options on the IVI driver installation screen. There is a button to select all instrument drivers, a button to deselect all instrument drivers, and a control to replace the existing drivers. This control can be set to either replace the instrument drivers currently installed with the IVI drivers, or to leave the existing instrument drivers. This is an important selection if you have made modifications to the current standard drivers; it will prevent the IVI installation from overwriting your changes.

The IVI installation will set up three categories of software. The installation categories are instrument drivers, utilities, and driver software. The instrument driver installation includes the IVI class drivers, the IVI class simulation drivers, and the IVI-specific drivers. The utility installation includes NI Spy, the Virtual Bench software, and the Measurement and Automation Explorer. The driver software includes the IVI engine, NI VISA, NI DAQ, and the CVI run-time engine. When the installation is complete, the computer will need to be restarted.

5.9.6 IVI Configuration

The first step, after installing the IVI software, is to run the IVI Configuration Utility. The IVI Configuration Utility can be started by double-clicking the Measurement & Automation icon on the Windows desktop, or by selecting the utility from the National Instruments IVI Driver Library folder in the Programs folder in the Start menu. The IVI configuration utility has a display similar to the Windows Explorer program. Figure 5.23 shows the IVI Configuration Utility Window.

There are four categories of IVI configuration items in the IVI folder. The main sections are Devices, Instrument Drivers, Logical Names, and Virtual Instruments. Inside the Virtual Instruments folder is a folder containing a Simulation Virtual Instruments folder. Inside the Instrument Drivers folder is the Class Drivers folder and the Simulation Drivers folder. Additionally, there is a utility for Creating a Logical Name in the main IVI folder.

To add an item to any of the IVI configuration folders, you can either right-click on the folder and select Insert or click on the Insert button that appears when you are inside a folder. When Insert is selected, an Insert Wizard launches. The wizard will walk you through the steps required to add an item to the folder you have selected. To edit the properties of an item in the IVI configuration files you can either right-click on the item and select the Properties option, or double-click on the item.

The configuration information for the IVI settings are stored in a file named “ivi.ini.” The default directory of this INI file is located in is the Vxipnp folder. By storing this information in an INI file, the ability to create multiple configurations becomes possible. The developer can create multiple INI files with the desired settings. The INI file can be opened from the IVI Configuration utility by selecting Open from the File menu. Here you are able to open or create a new INI file.
5.9.7 How to Use IVI Drivers

IVI class drivers are used in the same manner as standard instrument drivers. The IVI class drivers can be found in the Instrument I/O subpalette of the Functions palette. Each type of IVI class driver has its own subpalette. Each subpalette contains an Initialize and Close VI. There are also groups of VIs to perform instrument configuration, instrument functions, and utility functions that are necessary for the specific class driver. The developer can use these class drivers like typical VISA drivers. The programmer would put an Initialize VI on the diagram first. The main input to the Initialize VI is the logical name. The logical name is what tells the LabVIEW program what instrument and drivers to reference. As you will recall, in the setup of the IVI configuration items, the logical name references a particular virtual instrument. These logical names can be altered as needed using the Configuration utility. It is recommended that you set the name initially after installation and do not change it often. Applications that have been developed use this name, and may not work once the logical name has been altered. The virtual instrument refers to a specific driver in the Instrument Drivers folder, and a device. The specific driver then specifies the DLL containing the code module used to communicate with the device. The VIs associated with the instrument driver are placed in the Instrument Drivers palette during the installation.
The Initialize VI also has inputs to do an ID query and reset the instrument. The outputs of the VI are the Instrument Handle and the Error Out. The Instrument Handle can be passed throughout the VI and subVIs, just like a standard VISA instrument handle. Once the instrument is initialized, the functions required to perform the necessary programming task can be accomplished in two ways: the user can utilize the function VIs from the class driver subpalettes or make use of the LabVIEW Property node. When doing IVI driver configurations, the LabVIEW Property node is used in the same manner as ActiveX controls. As with all applications using communications, the final step is calling the Close IVI Class Driver VI. The following diagram shows an IVI example written with standard VIs and with the Property node. The VIs perform exactly the same function. Figure 5.24 illustrates the IVI example with and without the Property node.

### 5.9.8 IVI Virtual Bench

The IVI Virtual Bench VIs were designed to simulate the front panel of an instrument. The main use for the Virtual Bench VIs (or soft panels) is manual instrument control. The user can use the Front Panel VI to manually control the instrument. The

<table>
<thead>
<tr>
<th>TABLE 5.3</th>
<th>IVI Configuration File Types</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Folder</strong></td>
<td><strong>Description</strong></td>
</tr>
<tr>
<td>IVI</td>
<td>This folder contains the application used to create logical names. The remaining folders are also contained in this folder.</td>
</tr>
<tr>
<td>Devices</td>
<td>This folder contains a list of devices used with the defined instrument address included.</td>
</tr>
<tr>
<td>Instrument Drivers</td>
<td></td>
</tr>
<tr>
<td>Class Drivers</td>
<td>Contains the default class driver references. The class driver uses the default simulation virtual instrument when simulation is enabled, unless the configuration specifies a different simulation virtual instrument.</td>
</tr>
<tr>
<td>Simulation Drivers</td>
<td>Contains IVI simulation driver configurations. These configurations specify a code module (usually a DLL) that a class driver can use to simulate a device.</td>
</tr>
<tr>
<td>Logical Names</td>
<td>This folder contains the logical names. The logical names are used to reference a specific virtual instrument in the Virtual Instruments folder. The logical name is what is used to change which IVI driver is loaded when a new reference is opened.</td>
</tr>
<tr>
<td>Virtual Instruments</td>
<td>A virtual instrument is a reference to a specific driver in the Instrument Drivers folder and a specific device in the Devices folder. The properties configuration also has inputs for specifying attribute settings.</td>
</tr>
<tr>
<td>Simulation Virtual Instruments</td>
<td>The simulation virtual instrument refers to a simulation driver in the Simulation Drivers folder and supplies the initial attribute settings for the driver.</td>
</tr>
</tbody>
</table>
The similarity of the Virtual Bench VI to the actual instrument interface allows the user to be familiar with some of the function immediately. The key is when the specific type of instrument changes. If a Hewlett Packard oscilloscope is replaced with a Tektronix oscilloscope, the user should still be able to control the instrument with no noticeable change. Since the interface is the same, there are no new knobs or menus to learn. The IVI configuration files do all the work.

5.9.9 IVI Driver Example

The information above can become confusing. Every name seems to include either virtual, instrument, or driver. In addition, each type of file references one or more of the other IVI file types. In order to alleviate some of the confusion, an example will be provided to help clarify things. It should be noted that there are a few examples that come with the IVI library. The examples are contained in the following path: labview\examples\instr\iviClass.llb.

For this example we will be simulating an oscilloscope using the IVI drivers. The first step is to create a logical name. In the IVI Configuration utility we need to go to the Logical Name folder and select Insert. The Logical Name Wizard will prompt you for a name and description. For this example we will call the instrument “scope.” After typing “scope” into the name field and entering a description, you should click on Next. The Wizard will then prompt you for a virtual instrument to associate with the name. We will use an existing virtual instrument. The selections available in the pull down menu depend on the drivers you loaded when you installed IVI. The virtual instrument we selected was the HP54645-Hewlett Packard Oscilloscope. After you click on the Next button, the Wizard will prompt you for a device to associate with the virtual instrument. You can use an existing device, create a new device, or choose “None” (simulate the device). For this example we will simulate the device, so “None” was selected. The final step is to confirm the selections. You are free to go back to any previous step at any time prior to clicking on Finish.
If you go to the Logical Name folder, you will see the “Scope” name has been added. If you right-click on the name you can access the properties. On the summary page you can modify virtual instrument properties by clicking on the Properties button. You also have the ability to change the virtual instrument that this name refers to. This is where you would make the modification if an instrument were exchanged for another in your equipment rack. If the program were written properly, with only class drivers being used in the code, no modifications would be necessary to the application. In the Virtual Instrument tab of the Properties window, the programmer can change the specific driver or device. For example, if we were going to use a real oscilloscope, the device could be changed here.

The Inherent Attributes tab of the virtual instrument properties has two sections: Operations and Simulation. The Operation section allows the programmer to select options such as Range Checking and Query Instrument Status. The default is All Operations Selected. The Simulation section enables data simulation. The programmer has the choice of using the class driver or the specific driver for the output data simulation. We will be using the class driver for this example. The programmer also has the opportunity to either modify or change the simulation virtual instrument and its settings. If you click on the Properties button, the Simulation Virtual Instrument Properties window comes up. If you go to the Default tab, you will notice that the instrument default settings are configured here. You can go through the list of properties and change default values to match your instrument defaults. We will leave the current settings.

Going back briefly, the remaining two tabs of the Virtual Instrument Properties are the Channels and the Default Setup. The Channels tab allows the programmer to associate a virtual channel name with the specific channel string. The Default Setup tab allows the programmer to configure attribute settings outside of the application. The attribute settings selected here will be applied to the instrument after the Initialize function is called. This feature allows the programmer to set initial settings of instruments without code in the VI. When an instrument with different initial settings is used, the Default Setup can put the instrument into a known state without changing the application.

Now that the IVI configurations are set up, the application can be written. As has been mentioned before, only IVI class drivers should be used in the application to reduce the amount of modifications if a new instrument is used. This example is based on the IviScope example that comes with the IVI library (Acq Wfm Edge Triggered). The first coding step is to put the Initialize VI for the IVI oscilloscope class on the code diagram (IviScope Initialize.vi). For the logical name input, a string constant or control with the text “Scope” should be wired to the first terminal. The inputs for ID query and reset device are both defaulted as “true.” And, as always, the Error In cluster should be wired to the final terminal of the input connector.

Once the instrument is initialized, the inputs for the vertical and horizontal parameters need to be configured. In addition to the scope parameters, the triggering inputs need to be set. For this example, we will be using the Property node to configure the necessary parameters instead of the class drivers. The first code diagram is shown in the following figure. The first trial of this VI used one Property node to set all of the vertical, horizontal, and triggering parameters. Selecting the item from
either the ActiveX subpalette or the Application Control subpalette of the Function palette created the Property node. By using the positioning tool, you are able to increase the number of inputs by pulling the bottom corner down. The same task could also be accomplished by right-clicking on an input and selecting Add Element. A control was created for each input that was necessary with the appropriate default values being set. **Figure 5.25** displays the Scope Example Code Diagram 1.

After setting the chart and triggering parameters, the class VI to determine the actual number of data points to acquire. The output of this VI is then wired to the IviScope Read Waveform VI. The data from the Read Waveform VI is then bundled together and wired to the waveform graph on the front panel. Another minor change to the National Instruments example is the insertion of the IviScope Error Query VI. This is to illustrate the VXI Plug&Play Functions simulation. Finally, the Instrument Close VI is added to the code diagram.

Now it is time to debug our driver. The best method for debugging an application like this is to use the NI Spy utility to monitor the instrument communications. The NI Spy was discussed earlier in this chapter. There are a couple of IVI-specific items that need to be mentioned. When you go to the Spy menu of the NI Spy utility you will notice the installed IVI drivers available in the monitor list. For this example we will want to turn off the NI-VISA and the NI-488.2 monitoring options. They are being turned off to aid in interchangeability checking. If those items are turned off, any items with conflicts will be listed in blue. This will aid in spotting conflicts without having to go through all of the items captured. Once Capture has been turned on, we are ready to test our application.

When we press the **Start** button, the program starts executing. When the Configuration VI runs, the IviScope Simulator Setup screen appears. The IviScope Simulator Setup is shown in the following figure. There are a number of options available to you. The first option at the top of the window is View. There are three options to the view: Measurement Data simulation, VXI Plug&Play Functions

![FIGURE 5.25](image-url)
simulation, and Status Codes simulation. For this portion of the debugging we will be using the Measurement Data simulation only. Below the View selection is a checkbox for always prompting for output data. If the user wants this window to appear every time a measurement or status check is required, the checkbox should be selected. If the user only wants to set up the parameters at the beginning and let the program handle the rest, deselect the checkbox. For this driver simulation there are also selections for the parameters necessary for a waveform generator. This waveform will allow the program to receive measurable data. The IviScope Simulator is shown in Figure 5.26.

For this testing we will leave the default sine wave and deselect the checkbox for prompting when data is required. As you should now be aware, there is an error in the application. A message box comes up stating that an error occurred at the tenth argument of the Property node. The listed possible reasons are that a Null is required for the channel name when setting an attribute that is not channel-based. The message box also lists the bad attribute. Clicking Continue will close the message box and complete the execution. Since the horizontal parameters and the triggering parameters are not channel-based, they cannot be on the same property node as the vertical parameters. Figure 5.27 shows the modified code diagram that corrects this programming error.

Before we move on with the testing, let’s take a look at the API captures from the NI Spy utility. The NI Spy display is shown in Figure 5.28. The tenth entry in the list is the attempt to write the horizontal time per record parameter to the scope. Because an error occurred in this step, the line is in a red font. Double-clicking on the line opens up the Properties box. In the Properties box you can see what the inputs and outputs of the communications were. If you click on the Output tab, you will see the instrument returned the following error statement: (IVI_ERROR_CHANNEL_NAME_NOT_ALLOWED). This is followed by a description in the text box below the error statement.

Now that we have modified our VI, we can attempt to run the application again. This time we will again deselect the Prompting for Output Data checkbox. Before clicking “OK,” change the view to the VXI Plug&Play Function Simulation selection. In the section labeled “Pass the results you want the error query function to return” we will inject an error code and message. By doing this in a real application, you can test your codes ability to handle specific errors. This is especially important when doing error recovery in a state machine. This tool will allow you to test the error paths to ensure that proper actions are taken. After pressing “OK” you should be able to verify that the program returned the error code and message. In addition, the desired sine wave is displayed on the front panel.

The final step in the debugging process is verifying the communications in the NI Spy program. When you open the Capture window you should notice something different. One of the API calls is in a blue font. This is indicating that there is an interchangeability warning. If you double-click on the item to open the Properties window, you will see the Interchange Warnings tab. Listed on this tab are warnings about certain parameters that were not set. The reason this may be an issue is the differences between instruments. Some instruments have different default values. In order to ensure an application will work with whatever instrument is inserted, some parameters should be set even if it is the current instrument’s default value.
FIGURE 5.26

FIGURE 5.27
Additional simulations can be accomplished using the third and final option of the IviScope Simulator. The third view is the Status Codes Simulation display. By default, the status codes are not simulated. If the checkbox is selected, the list of functions becomes visible. In the column next to the function list is the corresponding status macro. This is the response the simulated driver returns to the program when the corresponding function is executed. The default setting is ALL VI_SUCCESS with a status value of 0. In order to change the return value, you first need to select a function. After a function is selected you can select a status macro to assign to the function by using the Status Code Macro ring control. The new status macro you select along with the default status value will be passed to the program when the specified function is called. You can change the status value as well by using the Custom Status Code input. This can add even more error checking to your arsenal. Give it a try.

BIBLIOGRAPHY

G Programming Reference, National Instruments
Data Acquisition Basics Manual, National Instruments
IVI-1: Charter Document, IVI Foundation
IVI-5: IviDmm Class Specification, IVI Foundation
Developing COM/ActiveX Components with Visual Basic 6, Dan Appleman, SAMS, 1998.
Application Note No. AN006, Developing a LabVIEW Instrument Driver, National Instruments
Application Note No. AN087, Writing Win32 Dynamic Link Libraries (DLLs) and Calling Them from LabVIEW, National Instruments.
Application Note No. AN120, Using IVI Drivers to Simulate Your Instrumentation Hardware in LabVIEW and LabWindows/CVI, National Instruments.
Application Note No. AN121, Using IVI Drivers to Build Hardware-Independent Test Systems with LabVIEW and LabWindows/CVI, National Instruments.
Application Note No. AN122, Improving Test Performance through Instrument Driver State Management, National Instruments.