Developing a LabVIEW™ Instrument Driver

Noël Adorno

Introduction

LabVIEW, the graphical programming language that pioneered the concept of virtual instrumentation, has been an enabling technology in the hands of scientists and engineers for over a decade. As LabVIEW has grown in popularity, so has the proliferation of instrument drivers, the software modules designed to control programmable instruments. To aid in the development of these drivers, National Instruments has created standards for instrument driver structure, device management, instrument I/O, and error reporting. This application note describes these standards, as well as the purpose of a LabVIEW instrument driver, instrument driver components, and the integration of these components. In addition, this application note suggests a process for developing useful instrument drivers. While these recommendations are primarily intended for those developers who intend to submit drivers to the National Instruments LabVIEW Instrument Library, other users should find this information equally useful. This document presumes that you understand basic GPIB, Serial and/or VXI concepts and are familiar with the operation of LabVIEW. You should also be familiar with communication with VISA.

The LabVIEW Instrument Driver

Overview

An instrument driver is a set of software routines that control a programmable instrument. Each routine corresponds to a programmatic operation such as configuring, reading from, writing to, and triggering the instrument. Instrument drivers simplify instrument control and reduce test program development time by eliminating the need to learn the programming protocol for each instrument. The LabVIEW Instrument Library contains instrument drivers for a variety of programmable instrumentation, including GPIB, VXI, RS-232/422, and CAMAC instruments. Because instrument driver VIs contain high level functions with intuitive front panels, end users can quickly test and verify the remote capabilities of their instrument without the knowledge of device-specific syntax. The end user can easily create instrument control applications and systems by programmatically linking instrument driver VIs in their block diagram.

Common Misconceptions

A LabVIEW instrument driver VI is not a live interactive front panel for the instrument. While instrument driver VIs can be run interactively, they do not contain continuous loops that read input settings and send instrument commands in response to real-time dynamic user input. Instead they read the controls of the front panel, format and send command strings to the instrument, read the responses to instrument queries, and update front panel indicators *once* per VI execution. This is a very important concept; in order for an instrument driver VI to work programmatically, it must be constructed so that interactive operator input is *not required* for it to run to completion. This precludes the use of file or text dialogs or VI setup options that will pop-up the VI when called in order to prompt for user input.

Of course, many applications may require exactly this type of interactive panel. You may want to prompt users to enter desired instrument settings using a menu-driven interface. How do you get such behavior from a VI not designed to work this way? With a LabVIEW instrument driver, there are many options for optimizing VIs for interactive use. For quick testing, you can easily force an instrument driver VI to run in a continuous loop by opening its front panel and clicking the Continuous Run button, which makes the VI act like a soft front panel that controls the instrument in real-time. For applications, you can build a higher-level VI that contains the desired interactive interface and calls the instrument driver VI at the appropriate time (or times) in the diagram. Alternatively, you can modify the instrument driver VI by enabling the **Open Front Panel When Called** option from the **VI Setup** options and adding loops and cases in the diagram.

Strict interactive use does not add much value to a VI; after all, users can simply press buttons on many instrument panels instead of running the VI. Remember, when you are building your LabVIEW instrument driver that it must be able to work *programmatically* as well as interactively. Do not use dialogs or other means that prompt for user input. Always wire all controls and indicators to the connector pane and pass all data in and out of VIs through these connectors.

A LabVIEW instrument driver is not limited to controlling a single instrument. Many users want to control several identical instruments at the same time using the same instrument driver. Is this possible? Certainly, if the instrument driver is designed correctly. Instrument driver VIs, like all other LabVIEW VIs, are serially reusable. Therefore, by calling an initialize VI several times with different addresses and then passing reference handles (or VISA sessions) between VIs, you can use the instrument driver VIs to control more than one instrument in an application.

For an instrument driver VI to be reusable (or multi-instance), the data contained within it must not be shared between different uses of the same VI. Normally, this is not a problem due to the LabVIEW dataflow structure; input data is consumed by the VI and output data is generated with each execution of the VI. It is only when global data is maintained by the VI that problems with reuse occur. For example, global VIs by definition are VIs whose data is meant to be shared. Uninitialized shift registers, between VI executions, also store data that can be shared between multiple uses of the VI. Instrument drivers using these storage mechanisms cannot be multi-instance drivers because the data stored in the VI by one instrument can be unintentionally read or overwritten by a second instrument.

The instrument drivers in the National Instruments instrument library are multi-instance drivers. Therefore, if you wish to submit your drivers to the instrument library, your drivers must also be multi-instance drivers. Make sure your front panels include the instrument handle control (VISA Sessions), and wire these handles to all I/O subVIs in your diagram. Next, remove all uninitialized shift registers and global VIs from the driver. This will eliminate data storage within the driver and make it reusable for more than one instrument.

For more details on multi-instance, please refer to the *Advanced LabVIEW Instrument Driver Development Techniques* application note.

Instrument Driver Architecture

Modern GPIB and VXIbus instruments are characterized by increasingly larger numbers of functions and modes. With this added complexity, it is necessary to provide a consistent design model that will aid both instrument driver developers as well as end users who develop instrument control applications. To define a standard for instrument driver software design and development, it is necessary to use conceptual models around which the design specifications are written. An external interface model will show how the instrument driver interfaces to other software components in the system. Similarly, an internal design model will define how an instrument driver software module is organized internally.

Instrument Driver External Interface Model

An instrument driver consists of software modules, or VIs, that the user can call interactively as well as from a higher-level software application. The model in Figure 1 illustrates how the instrument driver interacts with the rest of the system.

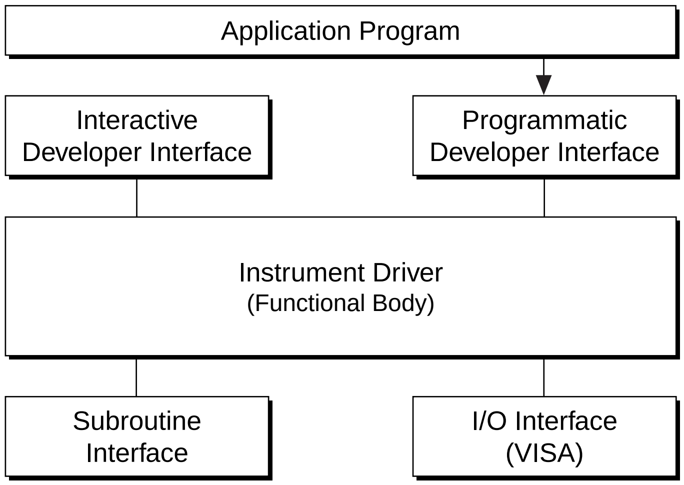


Figure 1. LabVIEW Instrument Driver External Design Model

This general model contains the instrument driver *functional body*, which is the code for the instrument driver. The details for the functional body are explained in the internal design model. *The programmatic developer interface* is the mechanism for calling the driver from a high-level application program. For example, a manufacturer's test system might make instrument driver calls to communicate with a multimeter or an oscilloscope. Therefore, the instrument driver sub-VIs would be used within a larger application. The *interactive developer interface* assists in the understanding of the function of each instrument driver VI. By running the front panels of the instrument driver sub-VIs, a developer can easily understand how to use the instrument driver in his application. The *VISA* (Virtual Instrument Software Architecture) *I/O interface* is the mechanism through which the driver communicates with the instrument hardware. VISA is an established standard instrumentation interface for controlling GPIB, VXI, serial, and other types of instruments from application software such as LabVIEW. The *subroutine interface* is the mechanism through which the driver can call support VIs that are needed to accomplish a task. For example, cleanup and error messaging VIs are considered necessary support VIs.

Instrument Driver Internal Design Model

To aid LabVIEW users in building their instrument control applications, National Instruments has developed libraries of instrument drivers for popular instruments. Each instrument driver has a number of VIs organized into a modular hierarchy containing not only high-level general-purpose application VIs, but also full-featured instrument driver component VIs.

The LabVIEW instrument driver internal design model, shown in Figure 2, defines the organization of the LabVIEW instrument driver *functional* body. This model is important to instrument driver developers because it is the foundation upon which the development guidelines are based. It is also important to end users because all LabVIEW instrument drivers are organized according to this model. Once you understand the model and how to use one instrument driver, you can use that knowledge for every LabVIEW instrument driver.

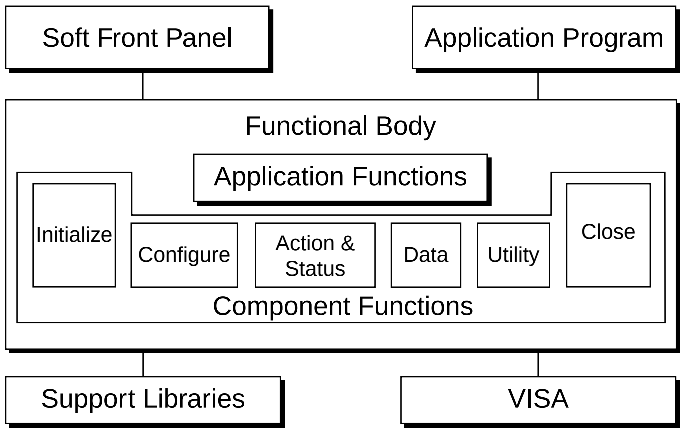


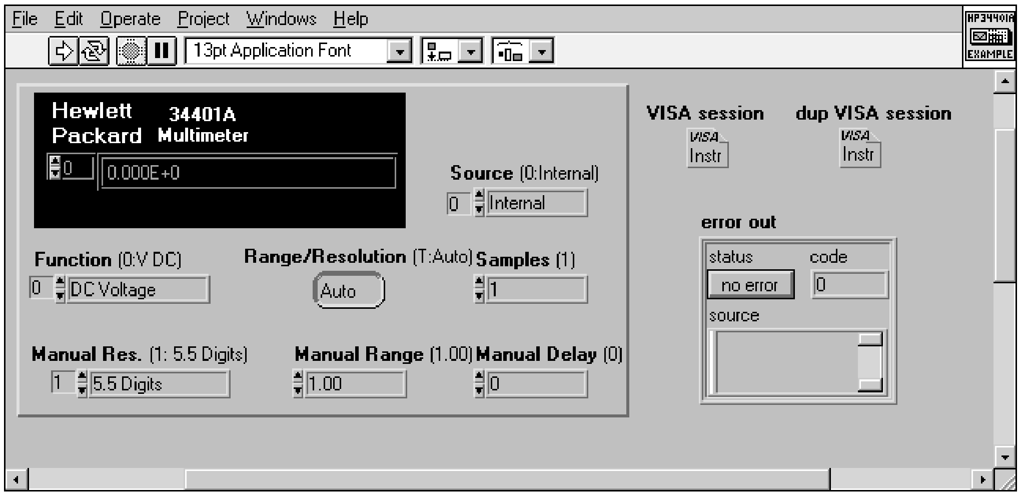
Figure 2. LabVIEW Instrument Driver Internal Design Model

The functional body of a LabVIEW instrument driver consists of two main categories of VIs. The first category is a collection of *component VIs*, which are individual software modules that each control a specific area of instrument functionality. The second category is a collection of higher-level *application VIs,* which combine component VIs to perform basic test and measurement operations with the instrument.

The internal design model of LabVIEW instrument drivers is built on a proven methodology. With this model, you have the necessary granularity to control instruments properly in your software applications. You can, for example, initialize all instruments once at the start, configure multiple instruments, and then trigger several instruments simultaneously. As another example, you can initialize and configure an instrument once, and then trigger and read from the instrument several times.

Instrument Driver Application VIs

The *Application VIs* are at the highest level of the instrument driver hierarchy. These high-level VIs are written in G block diagram source code and perform the most commonly used instrument configurations and measurements by calling the appropriate component-level VIs. They demonstrate high-level test and measurement functionality by configuring the instrument for a common mode of operation, triggering, and taking measurements. Because the application VIs are standard VIs with icons and connector panes, they can be called from any higher-level application requiring a single, measurement-oriented interface to the instrument. For some users, the application VIs are the only instrument driver VIs needed for instrument control. The HP34401A Application Example VI, shown in Figure 3, demonstrates an application VI front panel. The HP34401A instrument driver can be found in your **Function Palette** in **Instr»HP34401»Application VIs.**



0 0.000E+0

Source (0:Internal) 0 Internal

Function (0:V DC) 0 DC Voltage

Range/Resolution (T:Auto) Auto

Samples (1) 1

Manual Res. (1: 5.5 Digits) 1 5.5 Digits

Manual Range (1.00) 1.00

Manual Delay (0) 0

VISA session Instr

dup VISA session Instr

error out: status no error, code 0, source

Figure 3. HP34401A Application Front Panel.

Instrument Driver Component VIs

The application VIs are built from a lower level set of instrument driver functions called *component* VIs. Unlike the application VI (which presents only a subset of the instrument features), the component VIs are organized into a modular assortment containing all of the instrument configuration and measurement capabilities. The component VIs fit into six categories – *initialize, configuration, action/status, data, utility, and close.*

Initialize

All LabVIEW instrument drivers should have an initialize VI. It is the first instrument driver VI called and it establishes communication with the instrument. Optionally, it can perform an instrument identification query and reset operations. It may also place the instrument either in its default power on state or in some other specific state.

Configuration VIs

The configuration VIs are a collection of software routines that configure the instrument to perform the desired operation. There are usually a number of configuration VIs depending on the complexity of the instrument. After these VIs are called, the instrument is ready to take measurements or stimulate a system.

Action/Status VIs

The action/status category contains two types of VIs. Action VIs cause the instrument to initiate or terminate test and measurement operations. These operations can include arming the triggering system or generating a stimulus. These VIs are different from the configuration VIs because they do not change the instrument settings; they simply order the instrument to carry out an action based on its current configuration. The status VIs obtain the current status of the instrument or the status of pending operations. Although the specific routines in this category and the actual operations they perform are at the discretion of the developer, they usually are created on a need basis as required by other functions.

Data VIs

The data VIs include VIs to transfer data to or from the instrument. Examples include VIs for reading a measured value or waveform from a measurement instrument, VIs for downloading waveforms or digital patterns to a source instrument, and so on. The specific routines in this category depend on the instrument and are left up to the instrument driver developer.

Utility VIs

The utility VIs can perform a variety of operations that are auxiliary to the most often used instrument driver VIs. These VIs include the majority of the template instrument driver VIs (described below) such as reset, self-test, revision, and error query, and may include other custom routines such as calibration or storing/recalling instrument configurations.

Close

All LabVIEW instrument drivers should include a close VI. The close VI terminates the software connection to the instrument and deallocates system resources.

Each of these categories, with the exception of initialize and close, contain several modular VIs. Most of the critical work in developing an instrument driver lies in the initial design and organization of the instrument driver component VIs. The specific routines in each category are further categorized as either template VIs or developer-specified VIs.

Template VIs, available from National Instruments, are complete instrument driver VIs that can easily be customized. These VIs perform common operations such as initialize, close, reset, self-test, and revision query. The template VIs contain modification instructions for their use in a specific instrument driver for a particular instrument. For more information, refer to the LabVIEW Instrument Driver Templates section.

The remainder of the VIs, known as developer specified VIs, perform the actual instrument operations as defined by the instrument driver developer. Although all instruments will have configuration VIs, some instruments can have a different number of configuration VIs depending on the unique capabilities of the instrument. Although the specific VIs you develop will depend on the unique capabilities of your instrument, you should adhere to the categories discussed earlier – configuration, action/status, data and utility.

Using the internal design model as described in Figure 2, you can easily combine instrument driver VIs to create application programs. In cases when the included application VI is not optimized for a specific application, users can create new virtual instruments tailored to suit their needs by combining the component VIs as necessary. Users can further optimize the component VIs by adding or removing controls from the panels and modifying the diagrams. Figure 4 shows how instrument driver component VIs for the HP 34401A digital multimeter are used programmatically in the diagram of the application VI, HP34401A Application Example.

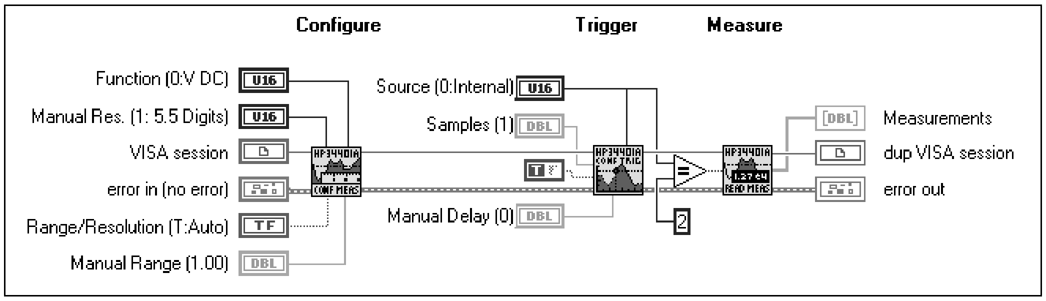


Figure 4. HP34401A Application Example Diagram.

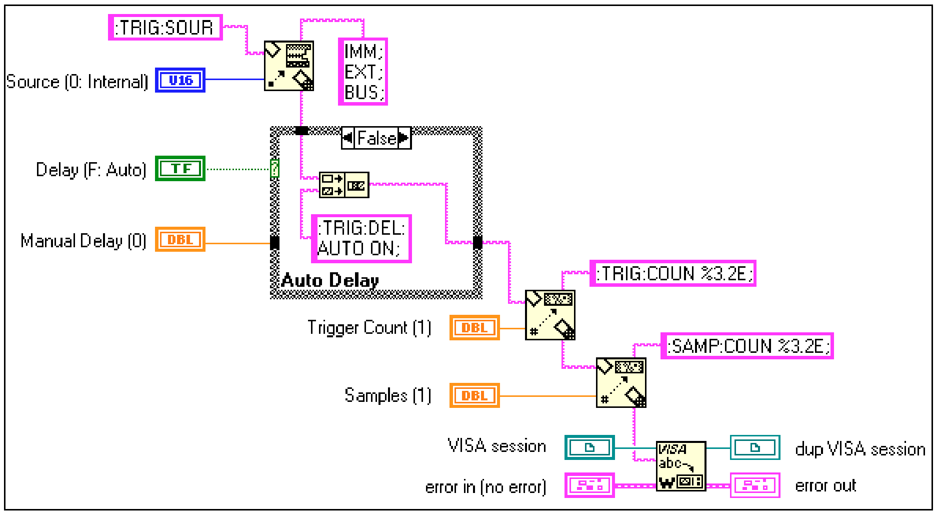


Figure 5. HP34401A Config Trigger Diagram

In the block diagram of the instrument driver component VIs, built-in LabVIEW primitives as well as VISA VIs are used to build command strings and send them to the instrument. The VISA VIs perform device management, standardized instrument I/O, and error handling. As shown in Figure 5, the command string is created by cascading formatting functions and then wiring the resulting string into the VISA Write VI. This VISA Write sends the command string to the instrument, checks for errors, and updates error cluster appropriately. The VISA VIs are discussed in more detail in the *VISA* section.

Additional VIs Distributed with the Instrument Driver

In addition to the VIs described by the internal model, an instrument driver should also include a Getting Started VI and a VI Tree VI.

Getting Started VI

Each instrument driver should contain a Getting Started VI. You can use this VI to interface with the instrument without wiring a subVI on the block diagram. This VI is usually the first VI the end user runs to verify communication with the instrument. This VI generally consists of three sub-VIs, the initialize VI, an application VI and the close VI. The front panel of the Getting Started VI resembles that of the application function it calls. Instead of having the user provide the VISA resource name, the user should have to provide only his/her GPIB address, VXI logical address, or communications port. For example, instead of having the user provide the name “`GPIB0::4`”, the Getting Started VI would require the user to supply only a GPIB address of 4. The front panel and block diagram of the Getting Started VI for the HP34401A are shown in Figures 6 and 7.

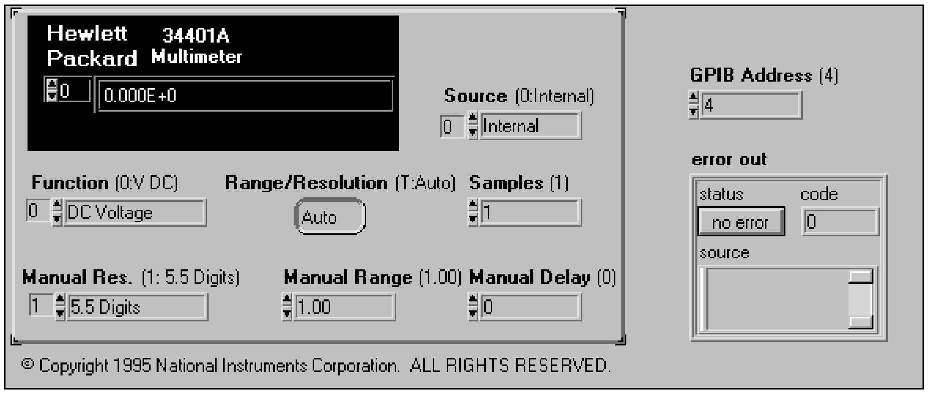


Figure 6. Front Panel of the HP34401A Getting Started VI.

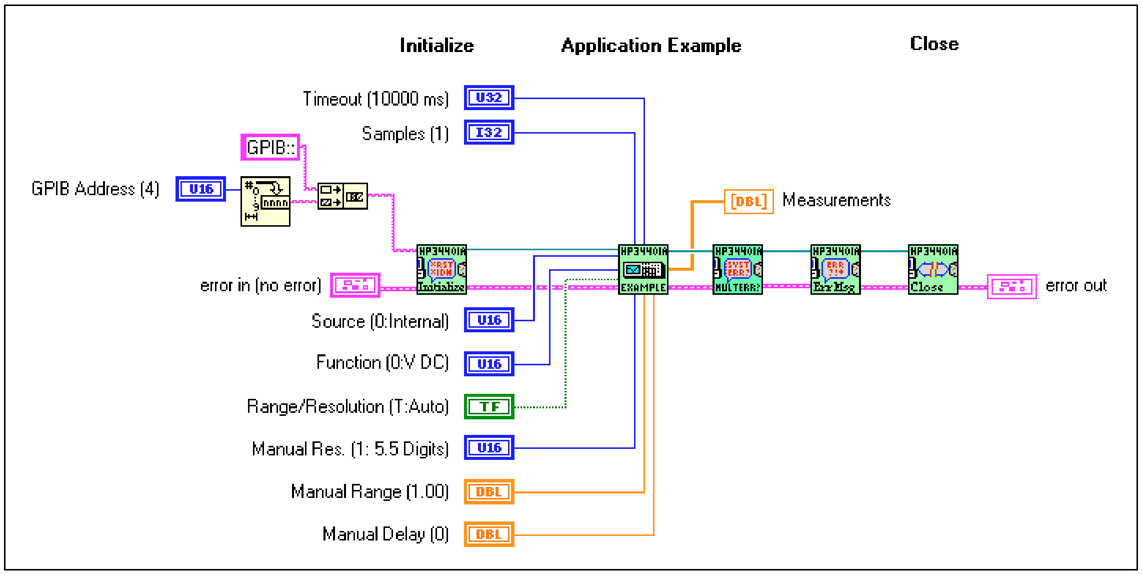


Figure 7. Block Diagram of the HP34401A Getting Started VI.

VI Tree VI

End users can view the entire instrument driver hierarchy at once with a VI Tree VI. This VI is a nonexecutable VI designed to show the functional structure of the instrument driver. If an end user does not install the palette menu files for the instrument, the VI Tree is the only resource to understanding the structure. An example of a VI tree VI is shown below in Figure 8.

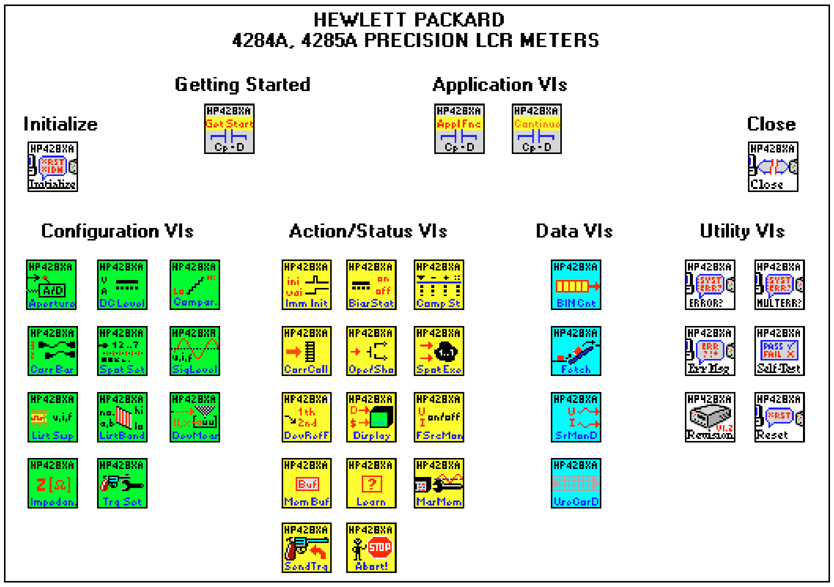


Figure 8. Block diagram of the HP428xA VI Tree VI.

LabVIEW Instrument Driver Development

This section describes the procedure for developing a LabVIEW instrument driver. The ideal LabVIEW instrument driver has full function control of the instrument. Rather than specify the required functionality of all instrument types, such as multimeters, counter/timers, and so on, this chapter focuses on the architectural guidelines of all drivers. With this information, driver developers can implement functionality unique to a particular instrument, and still organize, package and use all drivers in the same way.

The best way to develop a LabVIEW Instrument Driver is to follow a three-step process. In step one, you design the instrument driver structure. In step two, you modify the instrument driver templates VIs. In step three, you add developer-defined VIs.

Step 1. Designing the Instrument Driver Structure

The ideal instrument driver does what the user needs – no more and no less. No particular type of driver design is perfect for everyone, but by carefully studying the instrument and grouping controls into modular VIs, you can satisfy most users.

When the number of programmable controls in an instrument increases, so does the need for modular instrument driver design because a single VI cannot access all features. However, when an instrument driver contains hundreds of VIs, each controlling a single instrument feature, more instrument rules regarding command order and interaction apply. Modular design simplifies the tasks of controlling the instrument and modifying VIs to meet special requirements.

Ideally, you should devise the overall structure of your instrument driver before you build the individual VIs. A useful instrument driver is more than a series of VIs; it is a tool to help users develop application programs. You should design an instrument driver with the application and end user in mind.

You must create some instrument driver VIs that control unique instrument features. However, you can use template VIs for common operations. Template VIs are discussed in more detail in the Instrument Driver Template VIs section.

Instrument Driver Structure and VI Hierarchy

When you develop a LabVIEW instrument driver, it is important to define clearly the structure and VI hierarchy of the driver. First, define the primary VIs and develop a modular VI hierarchy. This hierarchy is the design document for a LabVIEW instrument driver.

Useful instrument drivers come from in-depth knowledge of the operation of the instrument and experience using it in real applications. The following steps outline one approach to developing the structure for a LabVIEW instrument drivers:

1. Familiarize yourself with the instrument operation. Read the operating manual thoroughly. Typically the foundation of the driver hierarchy is in the instrument programming manual. Learn how to use the instrument interactively before you attempt any programming.
2. Use the instrument in an actual test configuration to get practical experience. (The operating manual may explain how to set up a simple test.)
3. Study the programming section of the manual. Skim the instruction set to see which controls and functions are available and how the features are organized. Decide which features are best suited for programmatic use.
4. Examine existing instrument drivers for similar instruments. Often instruments from the same family have similar programming command sets that you can easily modify for your instrument.
5. Develop a structure for the driver by looking for controls that are used together to perform a single task or function. The sections of a well organized manual often correspond to the functional groupings of an instrument driver.

Instrument Driver VI Organization

After you have developed your Instrument Driver structure, you can develop a VI hierarchy to organize the VIs for the driver.

The VI organization of an instrument driver defines the hierarchy and overall relationship of the instrument driver component VIs.

You define the majority of instrument driver VIs and design them to access the unique capabilities of a particular instrument. However, many operations are common to all types of instrumentation. These common operations are performed by the template instrument driver Vis – initialize, close, reset, self-test, revision query, error query, and error message. You can find the template VIs in **Examples»instr»insttmpl.llb** VI library.

The template VIs for LabVIEW instrument drivers include ready-to-run VIs to perform these common instrument operations. The default command strings are based on the SCPI-compliant instruments. To include these VIs in your instrument driver, modify the command strings as required for your instrument. If the instrument is IEEE 488.2 compliant, few or no modifications are needed. If you are developing a driver for a non-IEEE 488.2 compliant or a register-based device, you will develop equivalent VIs for your instrument.

A class is a group of VIs that perform similar operations. Common classes of VIs are configuration, action/status, data, and utility.

The following table shows an example instrument driver organization for an oscilloscope. At the highest level of the hierarchy, you see the template VIs (initialize and close) and the typical classes of VIs.

Table 1. Organization Example for an Oscilloscope

VI Hierarchy	Type
Initialize VI	(Template)
Application VIs • Autosetup and Read Waveform • Rise-Time/Fall-Time Measurement	 (Developer Defined) (Developer Defined)
Configuration VIs • Configure Vertical • Configure Horizontal • Configure Trigger • Configure Acquisition Mode • Autosetup	 (Developer Defined) (Developer Defined) (Developer Defined) (Developer Defined) (Developer Defined)
Action VIs • Acquire Data	 (Developer Defined)
Data VIs • Read Waveform • Voltmeter Measurement • Counter/Timer Measurement	 (Developer Defined) (Developer Defined) (Developer Defined)
Utilities VIs • Reset • Self-Test • Revision Query • Error Query • Error Message	 (Template) (Template) (Template) (Template) (Template)
Close VI	(Template)

Guidelines and Recommendations

- Design an instrument driver VI front panel that contains all the controls required to perform the VI task.

  For example, a configure measurement VI would contain only the necessary controls to configure the instrument to take the measurement. It would not take the measurement or configure any other features. Other VIs in the instrument driver will perform these tasks.
- Design a modular instrument driver that contains a set of VIs, each performing a logical task or function, such as configuring the instrument or taking a measurement.

  A modular instrument driver is flexible and easy to use. For example, consider a digital multimeter driver design that uses a single VI both to configure the instrument and to read a measurement. The user cannot read multiple measurements without reconfiguring the meter each time the VI executes. A better approach is to build two VIs: one to configure the instrument, and one to read a measurement. Then the user can configure the meter once and take many measurements in a loop.
- Concentrate on the correct level of granularity in driver VIs and how these VIs will be used in a system.

  An instrument driver with a few very high-level VIs may not give the user enough control of the instrument operation. Conversely, an instrument driver with many low-level VIs is difficult for users unfamiliar with instrument rules regarding command order and interaction. For example, when using a measurement device such as an oscilloscope, the user typically configures the instrument once and takes many measurements. In this case, you should write high-level configuration VIs for the device. On the other hand, when using a stimulus device such as a pulse generator, the user may want to vary individual parameters of the pulse to test the boundary conditions of his system, or perform frequency response tests. In this case, you should write lower-level VIs, so that users can access individual instrument capabilities instead of reconfiguring each time they want to change one component of the output.

- Consider the relationship of the driver with other instrument drivers in the system.

  Typically, test designers want to initialize all of the instruments in a system at once, then configure them, take measurements, and finally close them at the end of the test. Good driver design includes logical division of operations.

- Create an instrument driver design (both in appearance and functional structure) that is similar to other instruments of the same type.

  Instrument drivers across a family of similar instruments should be consistent in appearance, structure, and style. For example, all oscilloscope drivers should resemble each other, as should all multimeters, scanners, and sources. If possible, modify a copy of an existing driver of a similar instrument.

- Design an instrument driver that optimizes the programming capability of the instrument.

  You can sometimes exclude documented functions that are not well suited for programmatic use. For example, most message-based devices have both a set and query version of each command. The set version is often needed for configuration of the instrument, but the query function is not needed. If the calls to set the instrument are successful, then the state of the instrument should be known.

- Design each VI to be independent of other VIs.

  If two or more VIs must always be used together, consolidate them into one VI.

- Minimize redundant parameters.

  For example, the parameters for each channel of a multichannel oscilloscope are similar or identical. Rather than duplicate the programming controls for each channel, you can include a VI control for selecting which channel to configure. The user can use this VI to change the settings for an individual channel, rather than configuring every channel each time the VI is called.

Design Example

Deciding which parameters to include in an instrument driver VI is one of the greatest challenges facing the instrument driver developer. Fortunately, organizational information is often available in the instrument manuals. In particular, the programming section of the manual may group the commands into sections such as configuring a measurement, triggering, reading measurements, and so on. These groupings can serve as a model for the driver hierarchy. Begin to develop a structure for the driver by looking for controls that are used together to perform a single task or function. A modular driver will contain individual VIs for each of the control groups.

A modular driver will also contain individual subVIs for each of the functions. Table 2 shows how the command summary from the Hewlett-Packard Digital Multimeter Operating Manual relates to developer specified instrument driver VIs.

Table 2. Comparison of Manual Sections with Instrument Driver VIs.

Virtual Instrument	Instrument Manual Section
HP34401A Initialize	**Input/Output Configuration** *IDN? *RST
HP34401A Config Measurement	**Measurement Configuration** AC filter Autozero Function Input resistance Integration time Range Resolution
HP34401A Config Trigger	**Triggering Operations** Reading hold threshold Samples per trigger Trigger delay Trigger source
HP34401A Config Math	**Math Operations** Math state, function Math registers
HP34401A Read Measurement	**Measurement Reading** Using Init and Fetch
HP34401A System Controls	**System-Related Operations** Beeper modes Display modes

While the instrument manual can provide a great deal of information about how to structure the instrument driver, you should not rely on it exclusively. Your knowledge of the instrument and how it is used should be the ultimate guide. The preceding table shows manual sections that map nicely to VIs found in the instrument driver. There are instances when it is more appropriate to place commands from several different command groups in your VI.

Conversely, it is often necessary to take one group of commands and divide it into two or more VIs. Consider how an instrument manual groups the trigger configuration commands with the commands that actually perform the trigger arming and execution. In this case, you should separate the commands into two VIs; one to configure the trigger, and one that arms or triggers the instrument.

Step 2. Modifying the Instrument Driver Templates

After you design the LabVIEW instrument driver structure, the next step is to modify the template VIs to represent your instrument. Most of the modifications involve the instrument prefix. The prefix is a unique identifier for the instrument driver, and is used as the filename for all files associated with the driver and as the prefix to all instrument VI names. Typically, the prefix is the combination of an abbreviation for the instrument vendor name and the model number. For example, the instrument prefix for the Tektronix VX4790 instrument driver is `tkvx4790`. As a default, the template instrument drivers use PREFIX as the instrument prefix.

Use the following procedure for modifying the LabVIEW instrument driver template:

1. Open the PREFIX Initialize template in the file `CoreDrv.llb` found in your `LabVIEW/examples/instr/insttmpl.llb` library.
2. Save the VI into a new VI library file by using the prefix for your instrument as the filename of the `.llb` file. Save the VI replacing PREFIX in the VI name with the prefix for your instrument.
3. Follow the instructions in the Modification Instructions string control on the Initialize front panel to modify the VI for your particular instrument.
4. Edit all Show VI Info... and control and indicator descriptions.
5. Edit the icon. Create an icon for each of the color modes of the icon: Black and White, 16-Color, and 256-Color.
6. Delete the Modification Instructions string control after you have completed the modifications.
7. Resize the front panel and save the VI.
8. Repeat steps 1 through 7 for PREFIX Close VI and the remaining template VIs that your instrument uses. All LabVIEW instrument drivers should have initialize, close, reset, revision query, error message, self test and error query and error message (multiple) VIs. If the instrument cannot perform some of the utility functions, the VI should return a “not supported” warning. Refer to the *Error Reporting* section for proper error and warning codes to be returned by the VIs.

After completing this procedure, you have a base-level driver that implements all template instrument driver VIs and is a good framework from which to create the rest of your driver.

In addition to `CoreDrv.llb`, there is one more instrument driver template library, `CoreDrU.llb`. This library should contain support VIs that the instrument driver uses internally, but are not intended for the end user to call. Two examples of support files, PREFIX Utility Clean Up Initialize and PREFIX Utility Default Instrument Setup, are included in the `CoreDrU.llb` file. If you intend the instrument driver to use these files, you should rename and modify them like those in `CoreDrv.llb`. For a description of each template VI, refer to the section on Instrument Driver Template VIs.

Step 3. Adding Instrument Driver Component VIs

The final step in developing a LabVIEW instrument driver is to add the developer-defined component VIs that define the functionality of the instrument driver and access the unique capabilities of your instrument. The VIs you create will be added to the source code along with the template VIs in the file `prefix.llb`, where prefix refers to your instrument driver prefix. For design and style details refer to the section on Details for Building Your Instrument Driver VIs.

You can use the following procedure to add your new VIs:

1. Open either the PREFIX Message-Based or PREFIX Register-Based templates VI in `CoreDrv.llb`. Use the PREFIX Message-Based template VI for message-based operations. Use the PREFIX Register-Based template VI for register-based operations.
2. Edit the VI front panel. Create the controls and indicators for the VI.
3. Edit all control and indicator Help information. Edit the **Show VI Info...** description.
4. Edit the icon. Create an icon for each of the color modes of the icon – Black and White, 16-Color, and 256-Color.

5. Edit the connector pane. Select an appropriate connector pattern and wire all controls and indicators to the terminals.
6. Edit the block diagram. Program all operations necessary to carry out the functionality of the instrument driver VI.
7. Save the VI.
8. Test the instrument driver VI.
9. Repeat these steps for every instrument driver component VI and application VI that you define for your instrument.
10. Edit the instrument driver `.llb` by selecting **File»Edit VI Library. . .** from the menu. Edit the Functions and Controls names. Edit the arrangement of icons in the Functions and Controls palettes by **selecting Edit Controls and Function Palettes. . .** from the **Edit** menu.

Editing the G block diagram source code is the most difficult step in adding a component VI to the instrument driver. Defining a block diagram structure makes it easier to edit the G source code. You can divide this process into the following steps:

1. Place the appropriate I/O routines in the block diagram.
2. Wire the **error in** cluster terminal to the first I/O VI error input connector. Then wire the error-out connector of that VI to the error-in connector of the next VI. Continue this process for all of the I/O VIs. Then wire the error-out connector of the last VI to the error-out terminal of the icon.
3. Wire the **VISA session** to every I/O VI. This is done in the same way as the error cluster.
4. Use the LabVIEW string VIs to assemble a command string based on the VI inputs.
5. Wire the command string to the VISA Write function.
6. Use the VISA Read function to read the response if the instrument generates a response.
7. Use the string VIs to parse the response and wire it to the appropriate indicator terminals.

Details for Building Your Instrument Driver VIs

This section differs from the last section in that it focuses on the three components of an instrument driver VI – the front panel, the block diagram and the icon. Each component will be discussed in detail for layout and style requirements.

The Front Panel

Once you decide which controls to group together to form an instrument driver subVI, you must decide which control styles best represent the instrument commands and options. Typically, instrument commands can be categorized into four types of control styles – Boolean, digital numeric, text ring numeric, or string. For example, any instrument command that has two options (e.g. `TRIG:MODE:AUTO|NORMAL`) can be represented on the front panel with a Boolean switch. In this case, you would label the switch Trigger Mode and add a free label showing the options: Auto or Normal. For commands that have a discrete number of options (such as `TRIG:COUP:AC|DC|HFREJ`), you should use a text ring rather than a digital numeric because the text ring can label each numeric value with the command it represents. Any command requiring a numeric parameter whose value varies over a wide range is better represented using a digital numeric rather than a long text ring. Finally, commands that require ASCII characters (such as a name) can be represented on a front panel with a string input control. These four control types – Boolean, numeric, text ring, and string input, are all you need to represent most instrument commands on the front panel of your VIs. The Simple Trigger VI in Figure 9 is an example front panel with the different type of controls.

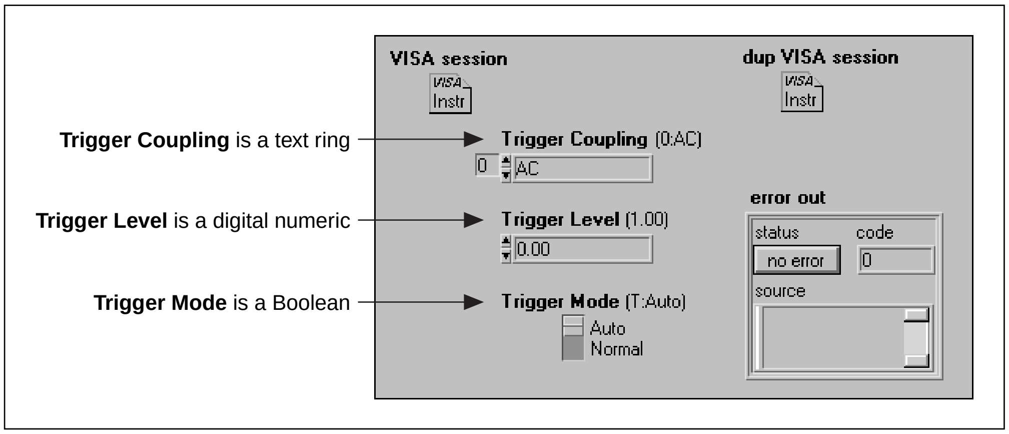
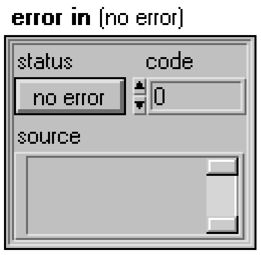


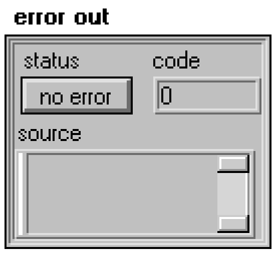
Figure 9. Simple Trigger VI

In addition to the controls required to operate the instrument, your front panel must also have the following required controls – VISA session, dup VISA session, error in and error-out. The VISA session handles are discussed in the Drivers Support Libraries section. The error clusters are described below:



error in describes error conditions that occur before this VI executes. The default input of this cluster is no error. The error-in cluster contains the following parameters:

- **status** is TRUE if an error occurred. If status is TRUE, this VI does not perform an operation. Instead, it passes the value of the error-in cluster straight to the error-out cluster.
- **code** is the error code associated with an error. A value of 0 means there is no error. Refer to the Error Codes section for a description of the possible error codes.
- **source** is the name of the VI that produced the error.



error out is a cluster containing error information. If error in contained an error, this is passed directly to error out. If error in did not indicate an error, the VI ran normally and error out describes any errors that may have been generated by the VI. This control should be placed in the lower right corner of your front panel.

When designing your front panels, use the following style guidelines to ensure uniformity with other LabVIEW VI front panels. First, use the default font (application) for all labels because this font is included with LabVIEW and thus is available to all other users. Also, use **bold** text to denote important or primary controls, and reserve plain text for secondary controls. (In most cases, all instrument driver controls are primary and require bold text.) Also, only the first letter of each word should be capitalized. The only exceptions are dup VISA session, error in, error out and acronyms which require caps (e.g. ID or GPIB). Enclose default information in parenthesis in the control name so the defaults can be seen in the help window when wiring into the VI. For example, you should label a function selector ring control whose default is DC volts at item zero **function** (`0:DCV`), and you should label a Boolean mode switch that defaults to true indicating automatic **trigger mode** (`T:Auto`); (Note, the default information is in plain text)**.** Place the VISA session control in the upper left, the dup VISA Session in the upper right and the error-out cluster in the lower right. Since the **error in** control is not designed to be used as an interactive input, place it on the left side off screen. Fill in all control descriptions as specified in the Online Help section. The Simple Trigger VI shown in Figure 9 is an example front panel which meets the style guidelines.

The Block Diagram

After designing your front panel, the next step is to create the G diagram that performs the function of the VI. As stated previously, each type of front panel control has a corresponding block diagram string function that simplifies the task of building command strings. Rather than wiring a Boolean control to the **Select** node and choosing a string constant to pass to a **Concatenate Strings** node, use **Select & Append**. This node both selects the proper string and concatenates it to the command string in one step. Likewise, use **Format & Append** to format and concatenate simple numeric values rather than using one of the **To Decimal** or **To Exponential** type conversions with the **Concatenate Strings** node. Again, **Format & Append** combines the functionality of the separate conversion and concatenate nodes and simplifies the block diagram considerably. For text rings, use **Select & Append**, and for string inputs use **Concatenate Strings**. You can find complete descriptions of these string functions in the *LabVIEW Users Manual.* The diagram in Figure 10 shows the preferred methods of building command strings.

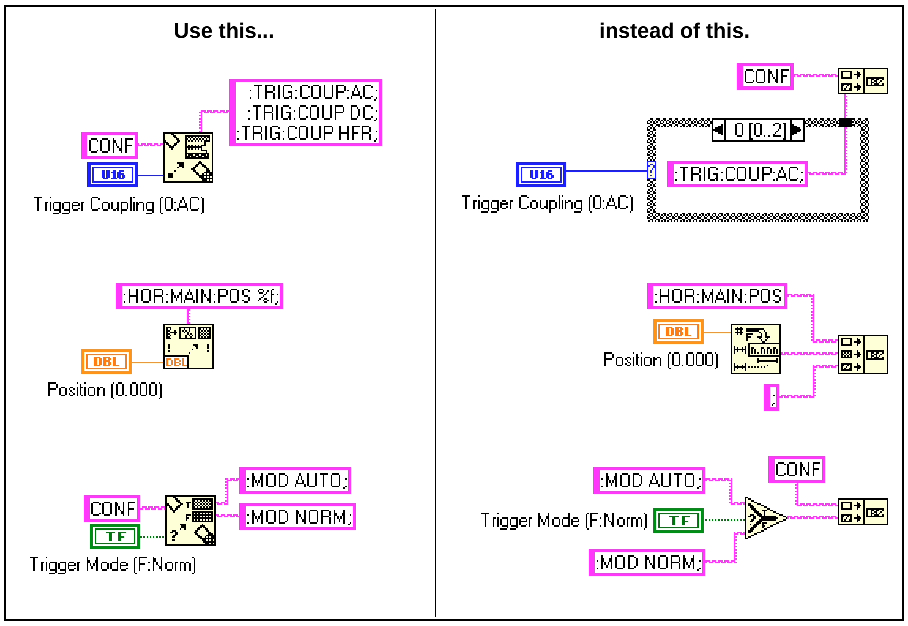


Figure 10. Comparison of Techniques for String Building.

Carefully consider the control flow while building your diagrams. LabVIEW does not necessarily execute in a left-to-right, top-to-bottom fashion. There is a great deal of data dependency that automatically determines execution order; add artificial data dependency wherever possible. You can use the error-in/out clusters to chain I/O functions together, thus defining the execution order without using case or sequence structures, as shown in Figure 4, HP34401A Application Example VI diagram. Although sequence structures also force the flow of execution, you should avoid them because they hide parts of your diagram, thus making it difficult for users to understand and modify.

Even with proper execution order defined, you may not know how much time is needed for the instrument to respond to commands. Timing problems can occur if the instrument driver VI attempts to send commands to the instrument while it is busy executing previously sent commands. Often, the new commands are ignored. Querying the instrument presents another potential problem. After commanding the device to send data, a period of time can elapse before the data is available. If you attempt to read during this period, data can be corrupted, a time-out can occur, or the instrument can malfunction. These internal timing problems can be overcome in a number of ways:

- Use events to signal that the instrument is ready to accept new commands or that data is available. If possible with your instrument, set bits in a service request mask register to configure specific SRQ events. Then you can use the

VISA Wait on Event function to suspend execution of the instrument driver VI until the device indicates that it is ready to continue. For more details on event handling, please refer to the *Advanced LabVIEW Instrument Driver Development Techniques* application note.

- Use status information to determine if the device is ready. You can query many instruments about their condition, and decode this information to determine if the desired condition exists before continuing the program.
- Insert appropriate time delays for instruments that cannot generate interrupts indicating that they are ready for new commands. Because most instruments have input buffers, it is usually possible to send a string containing several commands to the instrument. The individual commands are processed by the instrument one at a time, serially. Occasionally, an instrument requires a few moments to finish executing the commands in its buffer before it is ready to accept new commands or respond to a query. Only use the Wait (ms) function to impose a time delay when the instrument cannot be configured to generate a service request when ready, and status information is unavailable.

Along with these internal timing issues, you must also consider the interaction of the component VIs in your driver. If one component VI leaves the instrument in the wrong state, another component VI may not work properly. Additional timing problems may occur if one component VI sends commands to the instrument while the instrument is busy executing commands sent from another component VI. The techniques mentioned earlier are helpful in overcoming these problems.

Use proper wiring style to improve the diagram appearance and ease of understanding. Do not crowd the diagram; leave room for labels and wires. Do not cover wires with loops, case structures, labels, or other diagram objects. Also, reduce the number of bends in the wires by aligning data terminals whenever possible. You can use the cursor keys to move objects by single pixels if necessary. Use **Align and Distribute** in the toolbar to add symmetry and straight lines to your diagram. Also, add text labels to each frame of Case and Sequence structures. Alternatively, you can use enumerated types, which provide meaningful descriptions to cases within a case structure. You can label long wires and complex operations as necessary to increase clarity. Label control and indicator nodes with normal text, but use bold text to make your free label comments stand out. The background of your labels should be colored transparent. Refer to Figure 11 for an example of a block diagram which uses these diagram style techniques.

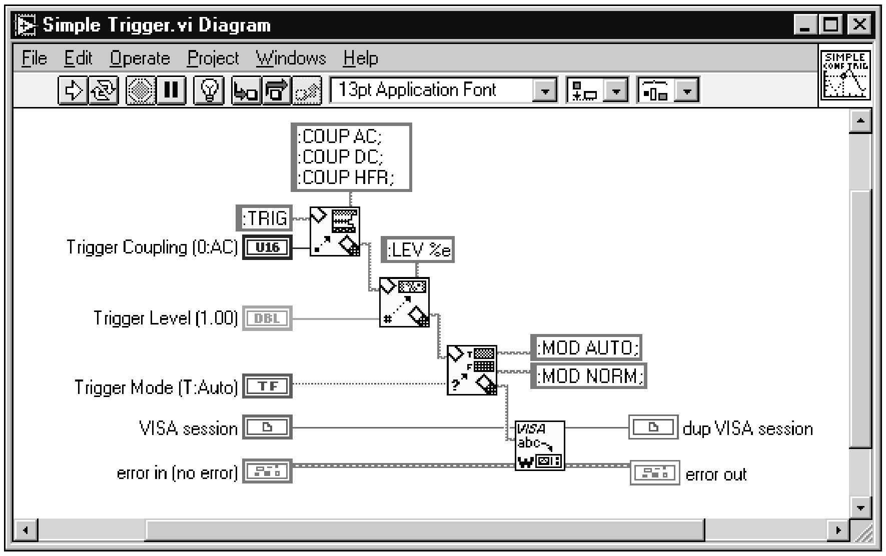


Figure 11. Simple Trigger Block Diagram Following Style Guidelines.

The Icon/Connector

There are a few rules regarding the VIs connector pane and icon. Reserve the upper left terminal for the VISA Session control, and the upper right terminal for dup VISA Session indicator. Reserve the lower left terminal for the error-in cluster and the lower right terminal for the error-out cluster to simplify wiring to subsequent error-in terminals. It is acceptable to choose a connector pattern that has extra terminals in case you have unforeseen control or indicator additions to your instrument driver VIs. This precaution prevents you from having to change the pattern and replace all instances of calls to a modified subVI. Place inputs on the left and outputs on the right whenever possible to promote a left-to-right data flow in the block diagram. Use meaningful icons for every VI. You can borrow icons from similar functions in other instrument drivers or use the icon library (`instricon.llb`) found in your LabVIEW/examples /Instr directory. Be sure to include text in the icon containing the instrument model controlled by the VI. If you are unable to create an icon to express the function of the VI, you can use text. Examples of icons are shown in Figure 12.

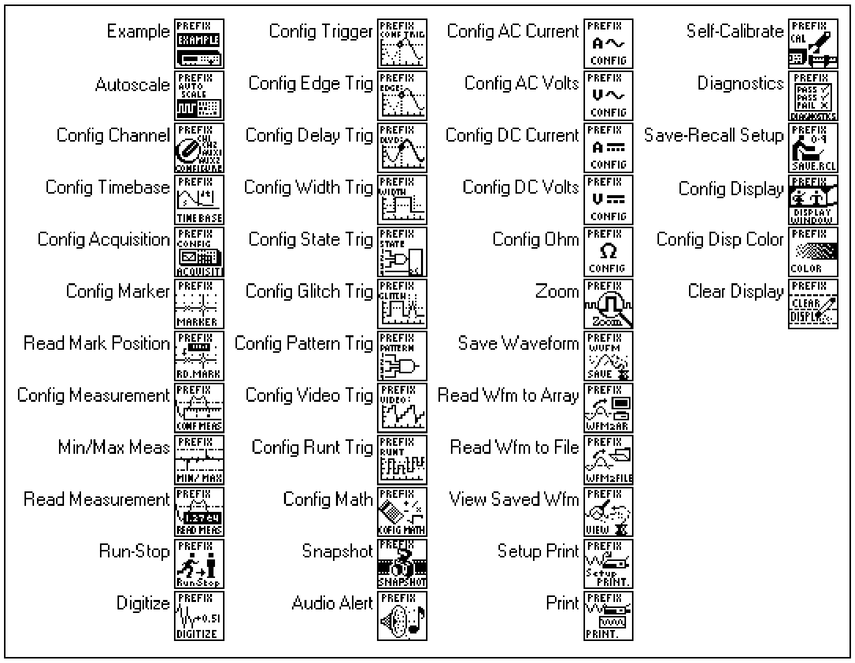


Figure 12. Sample Icons.

Important Considerations

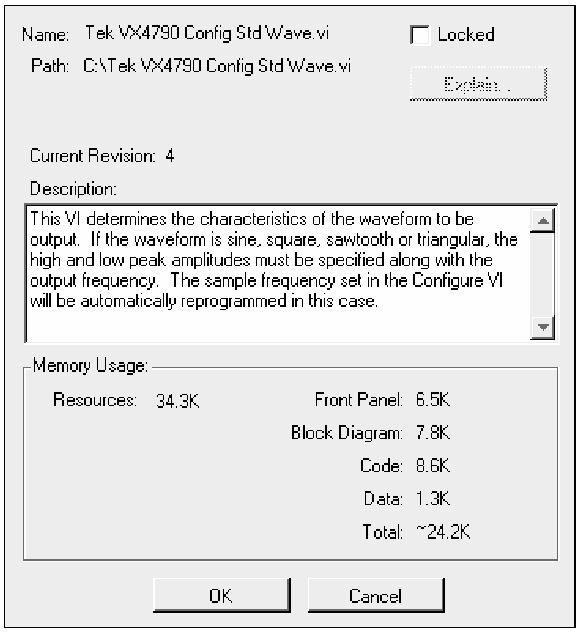
Application VIs

The application VIs demonstrate a common use of the instrument and show how the component VIs are used programmatically to perform a task. For example, an oscilloscope application VI would configure the vertical and horizontal amplifiers, trigger the instrument, acquire a waveform, and report errors. Don’t try to make your example VIs perform every function found in your instrument driver component VIs. Instead, concentrate on building simple, quality examples that can serve as a general model for users. Build the top-level examples by calling component VIs; don’t reproduce their code in the diagram of the application VI. Finally, do not use the instrument driver initialize or close VIs within the application VI because this will make it less useful in higher-level applications.

Online Help Information

To aid the user, you must include help for each instrument driver. LabVIEW has two types of help mechanisms:

- General Description help is available from the description box of the information window when a user selects **Show VI Info. . .** from the **Windows** menu (see Figure 13). This dialog box should contain a general description of the instrument driver VI including any control usage rules or VI interaction that should be brought to the user's attention.



Name: Tek VX4790 Config Std Wave.vi

Path: C:\Tek VX4790 Config Std Wave.vi

Locked

Current Revision: 4

Description:

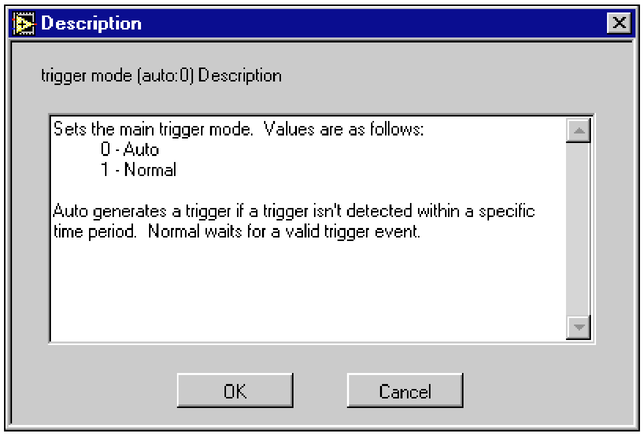
This VI determines the characteristics of the waveform to be output. If the waveform is sine, square, sawtooth or triangular, the high and low peak amplitudes must be specified along with the output frequency. The sample frequency set in the Configure VI will be automatically reprogrammed in this case.

Memory Usage:

Resources:	34.3K	Front Panel:	6.5K
		Block Diagram:	7.8K
		Code:	8.6K
		Data:	1.3K
		Total:	~24.2K

Figure 13. **Show VI Info…** Option from the **Windows** Menu.

- Control and Indicator help is the information most frequently viewed by the user. Provide a description of the individual controls and indicators. Front Panel Object help is obtained by selecting **Data Operations»Description. . .** from the object's pop-up menu (Figure 14). The help information contains the name and description of the parameter and its valid range and default value. Be sure to include information showing index numbers and corresponding settings for all ring and slide controls, settings corresponding to True/False positions on Boolean controls, and range information for numeric controls. You should also note any pertinent information concerning control interaction in the description boxes of each control affected by the interaction.



Description

trigger mode (auto:0) Description

Sets the main trigger mode. Values are as follows:
0 - Auto
1 - Normal

Auto generates a trigger if a trigger isn't detected within a specific time period. Normal waits for a valid trigger event.

Figure 14. Front Panel Object help from the **Description…** Option.

You can also help the user by placing free labels on the front panel and in the block diagram. In the block diagram especially, you should show all terminal labels (plain text) and color the borders transparent. Place free labels in case statements and sequence frames using bold text. This makes the comment stand out and makes the program easier to understand and modify. Also, using enumerated type ring controls rather than standard text rings can provide additional self-documentation to the block diagram.

Error Reporting

LabVIEW instrument drivers use the National Instruments standard for error reporting based on the use of a cluster to report all errors (Figure 15). Inside the cluster, a Boolean error indicator, a numeric error code, and an error source string indicator report if there is an error, the specific error condition, and the source (name) of the VI in which the error occurred (additional comments may also be included). Each instrument driver VI has an error-in and an error-out terminal defined on its connector pane in the lower left and lower right terminals respectively. By wiring the error-out cluster of one VI to the error-in cluster of another VI, you can pass error information throughout your instrument driver that will propagate to the top level VI in your LabVIEW application.

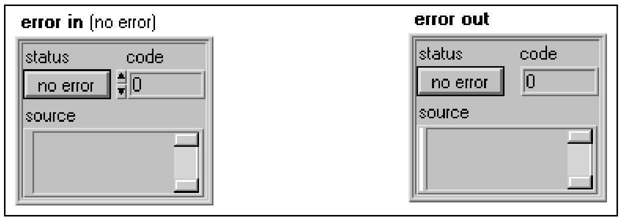


Figure 15. Error I/O Clusters.

A secondary benefit of error input/output is that data dependency is added to VIs that are not otherwise data dependent. This gives you a way to specify execution order beyond traditional sequence structures (see Figure 4). After the error cluster has passed through all of the instrument driver VIs, it can be passed to the Error Message VI (utility function within the instrument driver), General Error Handler VI or Simple Error Handler VI (found in the **Time and Dialog Function Palette** in LabVIEW) which interprets the error codes and displays messages to the user.

The VISA functions checks the Boolean state of the error-in cluster to determine if a previously executed VI generated an error. If an error is detected, the VISA function does not perform its usual operation. Instead it just passes the error information to the error-out cluster without modification. If no error is detected, the VISA function executes normally and determines whether it generated an error. If so, the new error information is passed to the error-out cluster, otherwise the error-in information is passed out. By using this technique, the first error triggers subsequent VIs not to execute (or some other action defined by the user) and the error code and the source of the error propagates to the top-level front panel. Additionally, warnings (error codes and source messages with the error Boolean set to false) will pass through without triggering error actions. For a list of VISA error codes, refer to the LabVIEW online help for error codes.

In addition to VISA error codes, there are several error and warning codes reserved for instrument drivers. These error codes are shown in Table 3. These codes should be returned by the instrument driver VIs when the appropriate condition occurs. You might see error codes like `–1300` for instrument specific errors in older instrument drivers and older instrument driver templates. In order to be more VXIPnP compliant the new codes should be used.

Table 3. Instrument Driver Error Codes

Hex Code	Decimal Code	Meaning	Generated by
0		No error: the call was successful	
3FFC0101	1073479937	WARNING: ID Query not supported	Instrument Driver
3FFC0102	1073479938	WARNING: Reset not supported	Instrument Driver
3FFC0103	1073479939	WARNING: Self-test not supported	Instrument Driver
3FFC0104	1073479940	WARNING: Error Query not supported	Instrument Driver
3FFC0105	1073479941	WARNING: Revision Query not supported	Instrument Driver
3FFC0800 to 3FFC0FFF	1073481728 to 1073483775	WARNING: Instrument specific warnings	Instrument Driver
BFFC0001	1074003967	ERROR: Parameter 1 out of range	Instrument Driver
BFFC0002	-1074003966	ERROR: Parameter 2 out of range	Instrument Driver
BFFC0003	-1074003965	ERROR: Parameter 3 out of range	Instrument Driver
BFFC0004	-1074003964	ERROR: Parameter 4 out of range	Instrument Driver
BFFC0005	-1074003963	ERROR: Parameter 5 out of range	Instrument Driver
BFFC0006	-1074003962	ERROR: Parameter 6 out of range	Instrument Driver
BFFC0007	-1074003961	ERROR: Parameter 7 out of range	Instrument Driver
BFFC0008	-1074003960	ERROR: Parameter 8 out of range	Instrument Driver
BFFC0010	-1074003952	ERROR: Interpreting instrument response	Instrument Driver
BFFC0011	-1074003951	ERROR: Identification query failed	Instrument Driver
BFFC0800	-1074001920	ERROR: Opening the specified file	Instrument Driver
BFFC0801	-1074001919	ERROR: Writing to the specified file	Instrument Driver
BFFC0803	-1074001917	ERROR: Interpreting the instrument's response	Instrument Driver
BFFC0804 to BFFC0FFF	-1073999873 to -1074001916	ERROR: Instrument specific errors	Instrument Driver

Prior to the introduction of error I/O clusters, LabVIEW instrument drivers had no consistent method for reporting error conditions. Additionally, invalid commands, syntax errors, or out-of-range values often caused early GPIB instruments to lock-up. For these reasons, error handling strategies focused on preventing sending strings to the instrument that would cause instrument failure. Front panel data coercion and block diagram techniques were often employed to automatically detect and correct potential error situations, usually without the knowledge of the user, who received no indication that his inputs were being overridden. Because newer instruments are capable of handling and reporting these situations, and because LabVIEW instrument drivers now have a consistent error reporting mechanism, the emphasis is shifting towards minimal error handling routines in the driver VIs, and using the error handling capabilities of the instrument to find and report errors. Earlier error handling methods have not been invalidated; however, you must determine the appropriate amount of error handling required by your VIs, based on the need for speed, user-friendliness, and the features and behavior of the instrument. As developers, you are not relieved of your duties of providing error handling; rather, you have greater responsibilities for providing good information to users about their inputs, and you have more tools to choose from to accomplish the task. Since most instrument drivers developed today use the query method to report errors, it is discussed in detail below.

Query the Instrument

As defined by the SCPI standard, many newer instruments have an error/event queue, which stores errors and events as they are detected. This queue is first in, first out, with a minimum length of two messages. In the event of overflows, the least recent errors/events are retained, while the most recent error/event is replaced with a queue overflow message. The SCPI standard defines common error types, including command errors, execution errors, device-specific errors, and query errors. Each error is stored in the queue, with a unique error/event number, optional descriptor, and optional device-dependent information. By issuing the `:SYST:ERR?` command, SCPI instruments return one entry from the queue, which may be an error, an overflow warning, or the message, ‘0, “No error’”. In your instrument driver application VIs, you can use this queue to detect and report instrument errors by querying the instrument after commands are sent. Obviously, querying the instrument for errors adds to the execution time of the VI, but this technique is beneficial as a “catch-all” mechanism for detecting instrument-specific errors. The only downside to this method is it requires the end user to use the Error Query VI within his application. Some end users do not implement error checking in their applications.

The Instrument Driver Templates VI library contains two versions of SCPI error reader VIs, which you can copy into your instrument drivers. The PREFIX Error Query (multiple) VI is the most useful with SCPI instruments since it flushes the instrument’s error buffer and detects the presence of any error messages. If errors are detected, the PREFIX Error Query (multiple) VI updates the error cluster with error code `-1074001916` (Hex `BFFC0804`) , and places into the source message the name of the VI performing the error query, as well as the error information returned from the instrument. The LabVIEW error handler VIs identify error code `-1074001916` as an instrument-specific error and generates an appropriate error message. The front panel of the Error Query (Multiple) VI is shown in Figure 16.

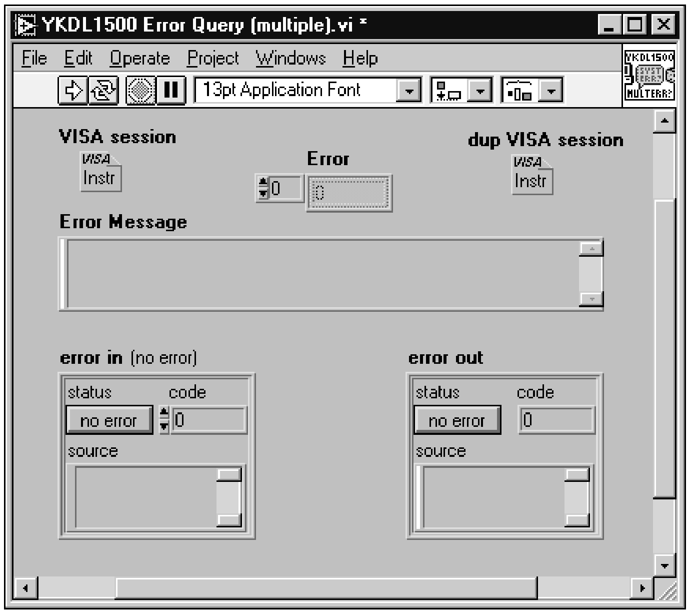


Figure 16. Example of the Error Query (multiple).vi.

To use the VI in your application, place it wherever you wish to query the instrument for errors. During initial development, you may want to place this VI in every instrument driver component VI to determine if your VI is generating instrument errors that could be prevented with a better algorithm. Remove them from the final version of the instrument driver to optimize the driver. In your application VI, call the Error Query VI after the component VIs have executed. Since SCPI instruments buffer the errors in a queue, the error query VI will be able to report all errors that were created throughout the application. If the error information returned from the instrument is detailed enough to determine exactly what went wrong in the instrument driver, you do not need to add extra programmatic error checking into your diagrams; use the capabilities of the instrument for this. If, on the other hand, the returned error information is cryptic or too general to be of any practical value, you must add more error checking in your VIs to detect and/or correct the errors before they reach the instrument. You want to inform end users of instrument error conditions; the error query VI is another tool in your arsenal that you can employ to meet that goal.

While error querying is a very effective method for detecting and reporting errors, it has some limitations. First, not all instruments have SCPI-defined error queues. For these instruments, you will have to modify (or replace) the error query VI with one of your own design to accommodate the capabilities of your instrument. Second, some of the instrument messages may not be specific enough to be of any practical value. For example, the instrument may report only a generic ‘parameter error’ when it detects a value out of range; this is not especially helpful if your VI has 10 numeric controls on it. Finally, you must be careful about using the information in an instrument error queue. Is the information current, or is it stale information from some previous instrument operation? By flushing the buffer completely, as with the PREFIX Error Query (multiple).VI, you can be certain that no old information remains queued up which might be read at a later time and misinterpreted as occurring after when it actually happened.

Additional Style Tips

End users generally appreciate consistency between instrument drivers. Similarly, if the front panel and block diagram are simple with an easy-to-understand layout, they are less intimidated about modifying the code. Some users will want to modify the code to optimize it for their special needs. The following items are listed to benefit the end user:

- Except for error in and error out, avoid using cluster controls and indicators in your VIs. Passing cluster information between VIs makes the application more complex for the user, who will need to bundle and unbundle the information in the clusters. Even if the number of inputs is large, as in some configuration VIs that exceed the number of input terminals on the left, top, and bottom of the icon connector, one should still try to avoid using clusters. You should either reevaluate the grouping of the inputs for the VI or use some terminals on the right side of the icon connector. Clusters should be used only when there is a logical grouping of controls, such as the error cluster, which will be passed and used by several VIs.
- Use color sparingly in your instrument driver design. Although your development machine might have lots of color, the end users might be using the instrument driver in an industrial environment on a black and white monitor or VGA monitor with just 16 colors. Similarly, while the development machine might have a high resolution monitor, the application machine might only have a resolution of 640x480. During development you want to make sure your front panels and block diagrams are readable and fit on various platforms.
- Use bitmaps sparingly on your front panels of your instrument driver VIs. Remember that these VIs will be later used as subVIs in a final application. The user generally will not popup or display instrument driver panels in his application. Rather, he will create his own panels to be seen by the application operators.
- Set the panel order with interactive users in mind. When using the instrument driver VIs interactively, users might prefer to use the keyboard to tab between the input controls rather than using the mouse. Make sure the panel order is set with a logical tab order in mind.

- Use enumerated types instead of standard text ring controls. It is preferable to use enumerated types since selections for case structures are self-documenting when wired directly to a enumerated-type control or constant. When end-users are creating their higher level applications and use the “Create Constant” or “Create Control” feature introduced in LabVIEW 4.0, their applications will be more understandable with a enumerated type rather than a numeric constant. The example in Figure 17 illustrates the difference.

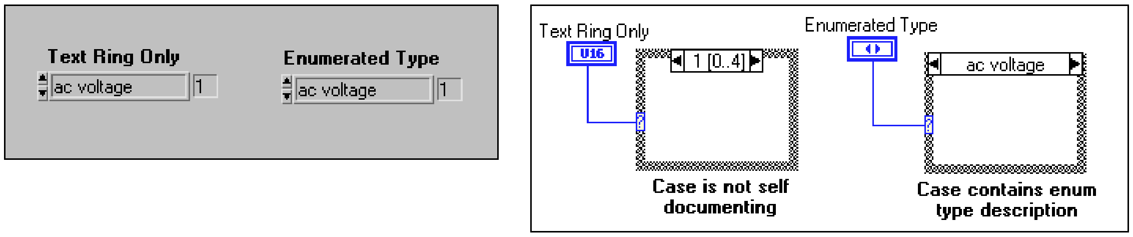


Figure 17. Comparison Between Using a Text Ring and an Enumerated Type Ring Control.

Do not crowd the diagram. Crowded diagrams and front panels are more difficult to understand than a simple and neatly organized VI. You will also want to give extra space around items with labels in order to account for font sizing differences when different printers or systems are used. You should also select “Size to Text” for all labels.

Creating Menus

In order to make it easier for customers to install, access, and use instrument drivers using LabVIEW version 4.0 and later, palette menus should be created and used. For consistency, instrument drivers should appear in the instrument library submenu of the function palette. Within the subpalette of the instrument VIs should have the same organization as the internal design model as shown in Figure 18. The initialize and close VIs should appear on each side of the application subpalette. The subpalettes for the component groups, configuration, action/status, data and utility should be on the second row of icons.

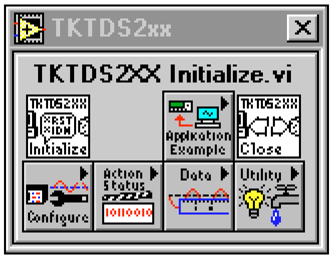


Figure 18. Example Subpalette for the HP34401A.

To achieve the same palette structure for all instrument drivers, it is best to start with the template menu files. Place the template menu files and your instrument driver files in a new folder within the **labview/instr.lib** directory. Re-launch LabVIEW – you should see the template VIs show up within the instrument drivers palette. Select **Edit Controls and Function Palettes. . .** from the **File** menu in LabVIEW. From the subpalette icon popup menu, you can make selections to edit the icon and change the name. Bring up the instrument driver subpalette window to view the hierarchy of the driver. For each subpalette, insert the VIs that correspond to that category. For instrument drivers with many sub-VIs, it might be easier to create a temporary subpalette that links to a complete VI library. Then, instead of inserting each VI in the subpalette, you can drag/copy the VIs from the temporary palette onto each component group subpalette.

Testing the Operation

It is a good idea to test your instrument driver as you develop it. Although most users will follow the online help to determine the inputs to the VIs, some will be confused and pass invalid data to the VI. Therefore, you should also test your VIs with invalid data, boundary conditions/ranges, and unusual combinations of inputs. Similarly, if string or array information is needed by a subVI, an empty array or empty string should be tested.

Driver Support Libraries

Overview

LabVIEW includes tools to aid in your instrument driver development. These tools include a library of template VIs that serve as a starting point for creating your own drivers, VISA functions to perform the instrument I/O, icon libraries to aid you in creating meaningful icons, and support files/functions. The main items of interest are the VISA functions and the Instrument Driver Template VIs.

VISA

The VISA functions, found in your **Instrument I/O->VISA** functions subpalette, contain the I/O interface used by instrument drivers to communicate with programmable instruments. VISA is a single interface library for controlling VXI, GPIB, serial, and other types of instruments.

Following are descriptions of the most commonly used VISA functions and controls. For more information, refer to the LabVIEW Function and VI Reference Manual.

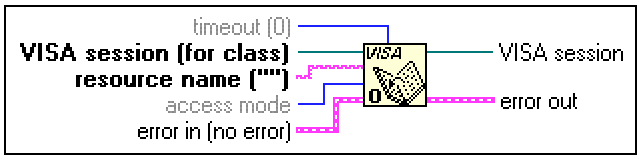
On the front panel of most instrument driver VIs is a **VISA session** control and a **dup VISA session** indicator. These controls and indicators provide a means of passing session information between sub-VIs. A **VISA session** is a unique logical identifier to a session. It is used to identify the resource being operated on by the VI. It is also used to differentiate between different sessions of the instrument driver applies.

VISA Session (except for the initialize VI) input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and passes all necessary configuration information required to perform the I/O. This control should be placed in the upper left corner of your front panel.

dup VISA session output contains the same identifier information as VISA session, but it passes the reference out of the VI and onto other subsequent VIs that will access the same instrument. Data dependencies are established when the VISA sessions are chained together. This control should be placed in the upper right corner of your front panel.

The **VISA session** controls are passed into and out of VISA function in the block diagram.

VISA Open



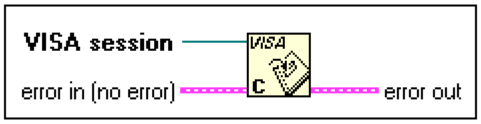
Establishes a communication session with a remote instrument based on the resource name. VISA Open creates a VISA session that is used by other VISA functions to perform operations on that session. Table 4 shows the syntax for the resource name. Optional string segments are shown in the square brackets ([ ]). Default values for optional parameters are as follows: board is 0; secondary address is none; GPIB-VXI primary address is 1.

Table 4. Instrument Description Syntax

Interface	Syntax
GPIB	`GPIB[board]::primary address[::secondary address][::INSTR]`
VXI	`VXI[board]::VXI logical address[::INSTR]`
Serial	`ASRL[board][::INSTR]`
GPIB-VXI	`GPIB-VXI[board]::VXI logical address[::INSTR]`

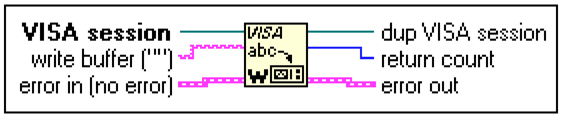
To interactively operate instrument drivers, you must first run this VI to generate the VISA session. Interactively you can access the VISA session in another VI by popping up the menu on the VISA session control and selecting an available session from the **Open Sessions. . .** sub menu. You can then run the instrument driver VI. Programmatically, simply pass a wire from the dup VISA Session indicator of the initialize VI to the VISA session control of the instrument driver VI you wish to run.

VISA Close



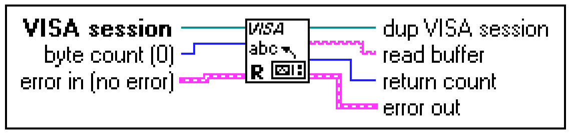
Closes the specified communication session with a device and deallocates system resources allocated to the instrument defined by the VISA session.

VISA Write



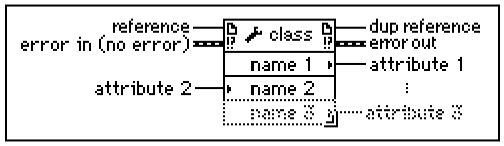
Writes data to a device. The data to be written is placed in the buffer represented by **write buffer**. The operation returns only when the transfer terminates.

VISA Read



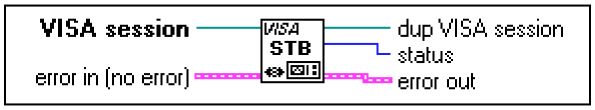
Reads data from a device. The data read is stored in the buffer represented by **read buffer**. The operation returns when the transfer terminates.

VISA Property Node



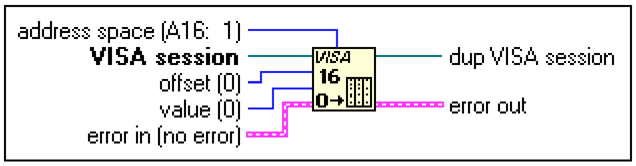
Gets and/or sets the named properties (attributes) for a given VISA session. Properties include such parameters as time-out settings, termination characters and manufacturer ID values.

VISA Read STB



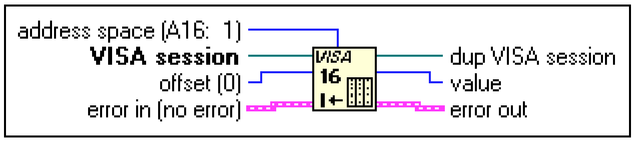
Reads the status byte from a message-based device.

VISA Move Out 16



Writes a 16-bit word to a VXI device at the specified address space and offset. This operation assumes supervisory data access and Motorola byte ordering.

VISA In 16



Reads a 16-bit word from a VXI device at the specified address space and offset. This operation assumes supervisory data access and Motorola byte ordering.

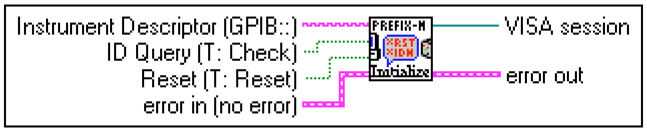
Instrument Driver Template VIs

The LabVIEW instrument driver templates, located in the `CoreDrv.llb` and `CoreDrU.llb` in your LabVIEW/examples/instr directory, contains a set of VIs common to most instruments. Since this template is updated periodically, you should download the latest version from National Instruments FTP site (ftp://ftp.natinst.com/support/labview/instruments/windows/current/cores/). You can use these VIs as a starting point for your instrument driver development. The templates have a simple, flexible structure and they establish a standard format for all LabVIEW drivers.

The front panels of the template VIs contain modification instructions on how to modify the VIs for a particular instrument. The template VIs are for use with both message-based instruments (GPIB, VXI and serial) as well as VXI register-based instruments. The instrument drivers can be used with IEEE 488.2 compatible instruments with minimal modification. For other instruments, you should use the template VIs as a shell or pattern for your VIs by substituting your instrument specific commands where applicable.

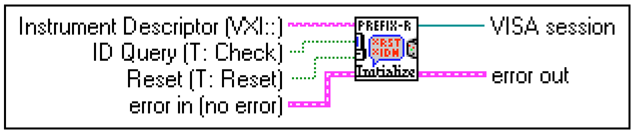
Following is a brief description of the instrument driver template VIs. For more information, consult the LabVIEW Function and VI reference manual.

PREFIX Initialize



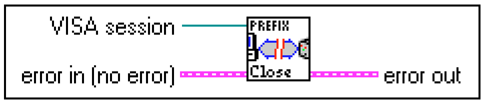
A template for creating an initialize VI for message-based instruments. It establishes communication with a remote instrument, and optionally performs ID query and/or reset operations (`CoreDrv.llb`).

PREFIX Initialize (VXI, reg-based)



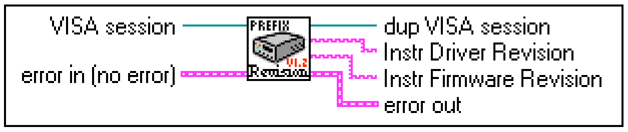
Similar to PREFIX Initialize, but used for VXI register-based instruments. It substitutes register reads for the string search when performing an ID query (`CoreDrv.llb`).

PREFIX Close



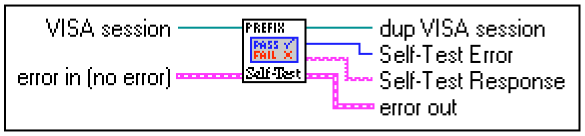
Terminates communication with the instrument and deallocates system resources (`CoreDrv.llb`).

PREFIX Revision Query



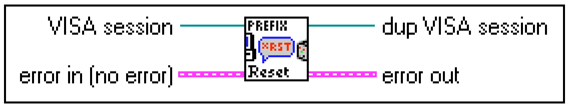
Returns the revision of the instrument driver and the firmware revision of the instrument (`CoreDrv.llb`).

PREFIX Self Test



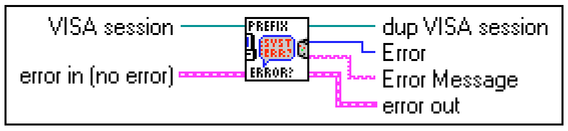
Instructs the instrument to perform a self-test and returns the result (`CoreDrv.llb`).

PREFIX Reset



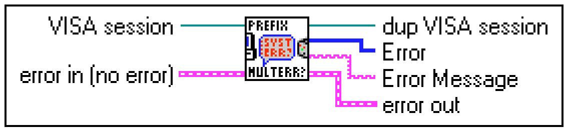
The Reset VI places the instrument in a default state (`CoreDrv.llb`).

PREFIX Error Query



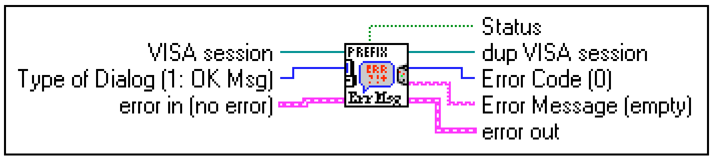
Queries the instrument once and returns instrument-specific error information. If an instrument error is detected, it is placed in the error-out cluster (`CoreDrv.llb`).

PREFIX Error Query (multiple)



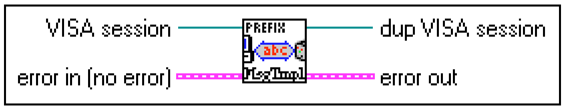
Similar to PREFIX Error Query, but continues to query in a loop until the error queue in the instrument has been cleared of all errors (`CoreDrv.llb`).

PREFIX Error Message



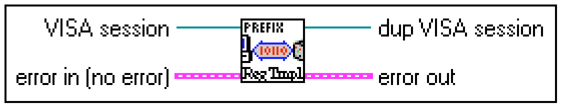
Calls the built-in error handler in LabVIEW to translate error status to a user-readable string (`CoreDrv.llb`).

PREFIX Message-Based Template



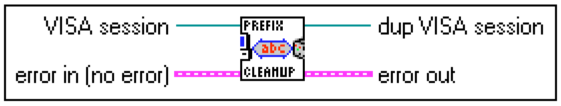
A template for creating a message-based component VI for your particular instrument. Calls the I/O VIs needed for message-based communication (`CoreDrv.llb`).

PREFIX Register-Based Template



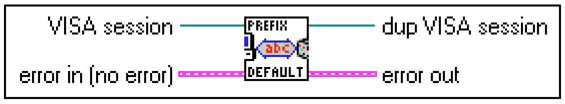
Similar to PREFIX Message-Based Template, but calls the low-level I/0 functions required for high-speed communication with a VXI register-based device (`CoreDrv.llb`).

PREFIX Utility Clean Up Initialize



This Utility VI closes an open VISA session in the event that there is an error during initialization. This VI is designed to be called only from the PREFIX Initialize VI (`CoreDrU.llb`).

PREFIX Utility Default Instrument Setup



This Utility VI sends a default command string to the instrument whenever a new VISA session is opened, or the instrument is reset. This VI is intended to be used as a subVI for the Initialize and Reset VIs (`CoreDrU.llb`).

PREFIX VI Tree

The VI Tree VI is a nonexecutable VI that is designed to show the functional structure of the instrument driver. It contains the Getting Started VI, application VIs and all of component VIs (`CoreDrU.llb`).

Final Comments

For the developer, defining the structure and constructing the VIs are the most important and time-consuming processes in the development of an instrument driver. The best instrument drivers group related instrument controls into modular VIs, each of which performs a task analogous to the way the instrument would actually be used. Ideally, with this type of structure, users will have on each individual panel exactly what they need to perform the particular instrument operation – no more and no less. The greatest challenge in developing instrument drivers lies in determining which controls belong on each particular VI.

For the user, the logical structure, help facilities, and error reporting are the most important features of the instrument driver. You must include appropriate comments in all description boxes, and you should document your code with comments in the diagrams. Build useful error reporting into your VIs by using the techniques described in this document. Thoroughly test all your VIs to ensure that they work properly.

Proper instrument driver development requires more than simply building and sending strings to instruments. Fortunately, the Instrument Library contains many fine examples of instrument drivers for a variety of instruments. Whether you are modifying an existing driver or developing a new driver from scratch, begin with the instrument driver template VIs. Not only do the templates contain VIs common to most instruments, but they also demonstrate the desired style and structure. From there, follow the internal design model and keep in mind the categories of component VIs as you build your functions. These proven tools will help you design instrument drivers that are acceptable to a wide range of users.

Additional and related references:

1. *LabVIEW Instrument Driver Checklist and Submittal Form*, Application Note 115, available on National Instruments FTP or web site (`www.natinst.com/idnet`).
2. *LabVIEW Instrument Driver Standards*, Application Note 111, available on National Instruments FTP or web site (`www.natinst.com/idnet`).
3. *Advanced LabVIEW Instrument Driver Techniques*, application note to be available by mid-1998,
4. *NI-VISA Programmer Reference Manual* (P/N 321073A-01) and *NI-VISA User Manual* (P/N 321074A-01)
5. *LabVIEW Function and VI Reference Manual* (shipped with LabVIEW).
6. *LabVIEW Style Guide*, available on FTP (`ftp.natinst.com/support/labview/documents/style-guide`).
7. *Instrument Driver Template VIs* (`CoreDrv.llb` *and* `CoreDrU.llb`), available on National Instruments FTP or web site (`www.natinst.com/idnet`).
8. *VXIPnP Specifications*, available on `www.vxipnp.org`
9. Other instrument drivers, available on `www.natinst.com/idnet`.

340018D-01 May98