Developing a LabVIEW Instrument Driver

Back to Document

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 more than 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 document describes these standards, as well as instrument driver components and the integration of these components. In addition, this document suggests a process for developing useful instrument drivers.

While these recommendations are primarily intended for 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 also should be familiar with communication with VISA.

Table of Contents:

Instrument Driver Architecture

Modern GPIB and VXI 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 aids 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 shows how the instrument driver interfaces to other software components in the system. Similarly, an internal design model defines 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. Figure 1 illustrates how the instrument driver interacts with the rest of the system.

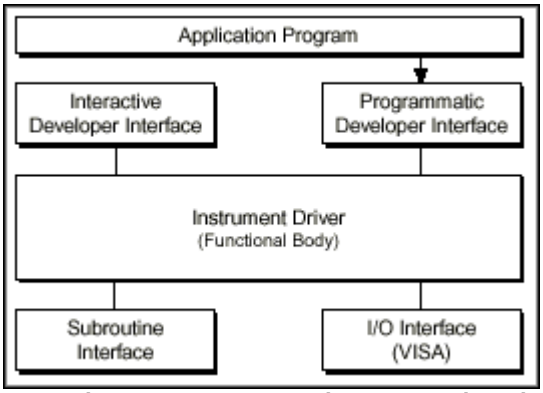

Figure 1. LabVIEW Instrument Driver External Design Model

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 test system might make instrument driver calls to communicate with a multimeter or an oscilloscope. Therefore, you would use the

instrument driver subVIs 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 subVIs, you can easily understand how to use the instrument driver in your application. The Virtual Instrument Software Architecture (VISA) 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. Refer to the *Driver Support Libraries* section of this document for more information about VISA. 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 also is important to end users because all LabVIEW instrument drivers are organized according to this model. After you understand the model and how to use one instrument driver, you can use that knowledge for every LabVIEW instrument driver.

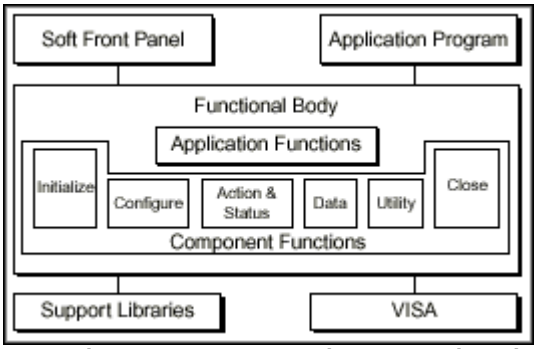

Figure 2. LabVIEW Instrument Driver Internal Design Model

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 applications. For example, you can initialize all instruments once at the start, configure multiple instruments, and then trigger several instruments simultaneously. Also, 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 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, you can call them 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 VIs are located on the **Functions** palette.

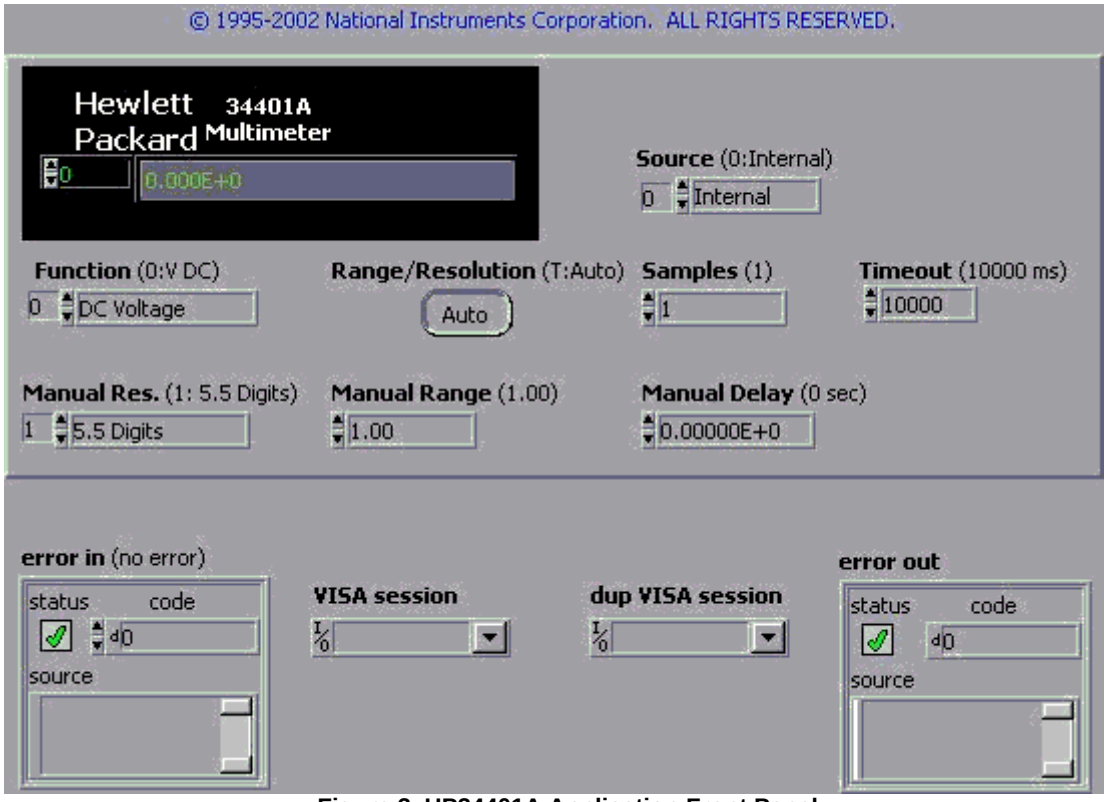

Figure 3. HP34401A Application Front Panel

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 that 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 also can place the instrument either in its default power on state or in some other specific state.

- **Configuration-** 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 you call these VIs, the instrument is ready to take measurements or stimulate a system.

- **Action/Status-** 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 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 --** The data VIs 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--** 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 such as reset, self-test, revision, and error query, and can 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 you can easily customize. 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. Refer to the *LabVIEW Instrument Driver Templates* section of this document for more information.

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 have configuration VIs, instruments have a different number of configuration VIs depending on the unique capabilities of the instrument. Although the specific VIs you develop depend on the unique capabilities of your instrument, you should adhere to the categories described earlier -- configuration, action/status, data, and utility.

Using the internal design model as shown in Figure 2, you can easily combine instrument driver VIs to create applications. In cases when the included application VI is not optimized for a specific application, you can create new VIs tailored to your needs by combining the component VIs as necessary. You can further optimize the component VIs by adding or removing controls from the front panels and modifying the block diagrams. Figure 4 shows how instrument driver component VIs for the HP34401A digital multimeter are used programmatically in the block diagram of the application VI, HP34401A Application Example.

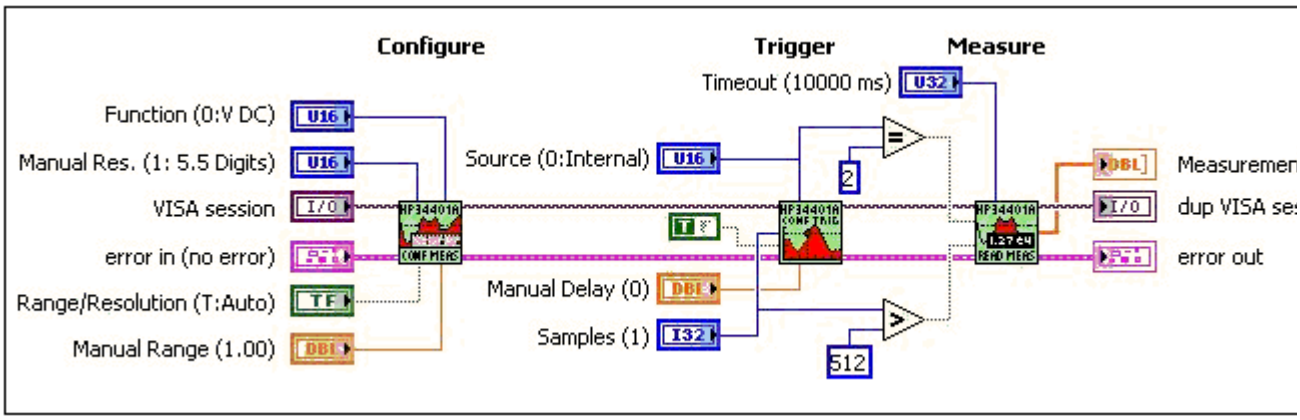

Figure 4. HP34401A Application Example Block Diagram

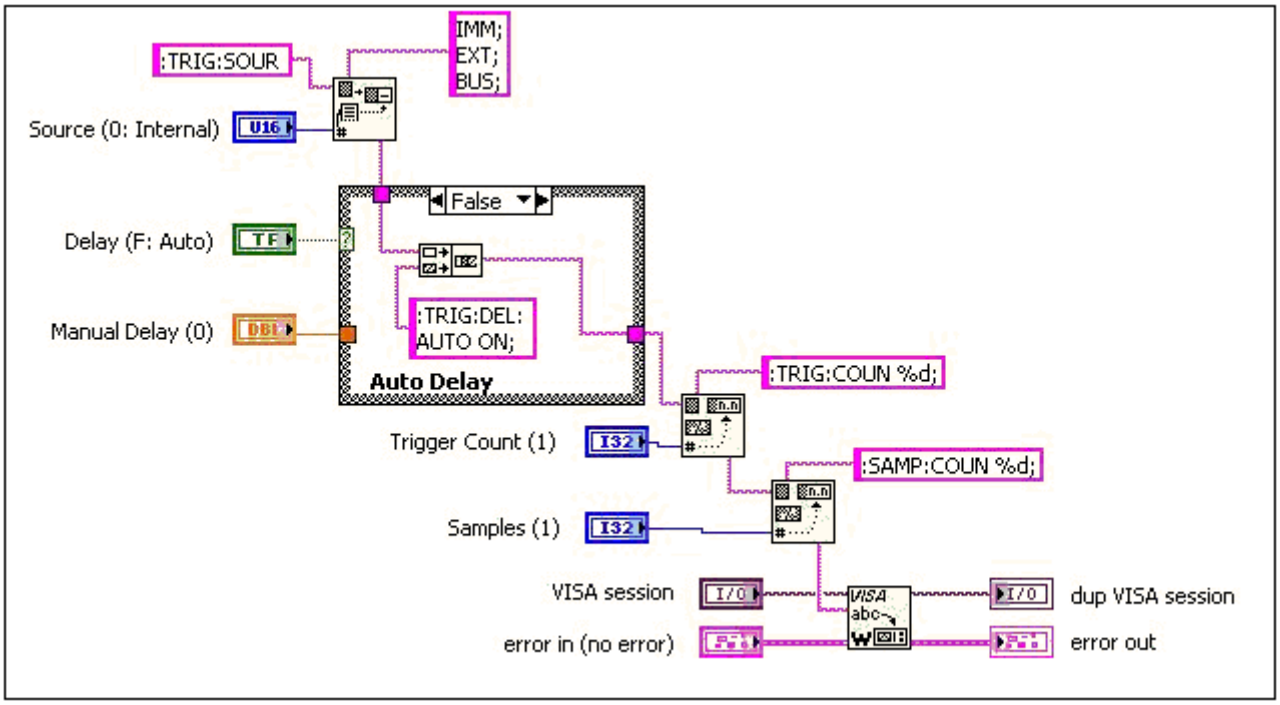

Figure 5. HP34401A Config Trigger Block Diagram

In the block diagram of the instrument driver component VIs, built-in LabVIEW functions as well as VISA functions build command strings and send them to the instrument. The VISA functions 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 function. This function sends the command string to the instrument, checks for errors, and updates error cluster appropriately. Refer to the *VISA* section of this document for more information about VISA functions.

Additional VIs Distributed with the Instrument Driver
In addition to the VIs described by the internal model, an instrument driver also should 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 subVIs -- 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 provide only the GPIB address, VXI logical address, or communications port. For example, instead of requiring 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.

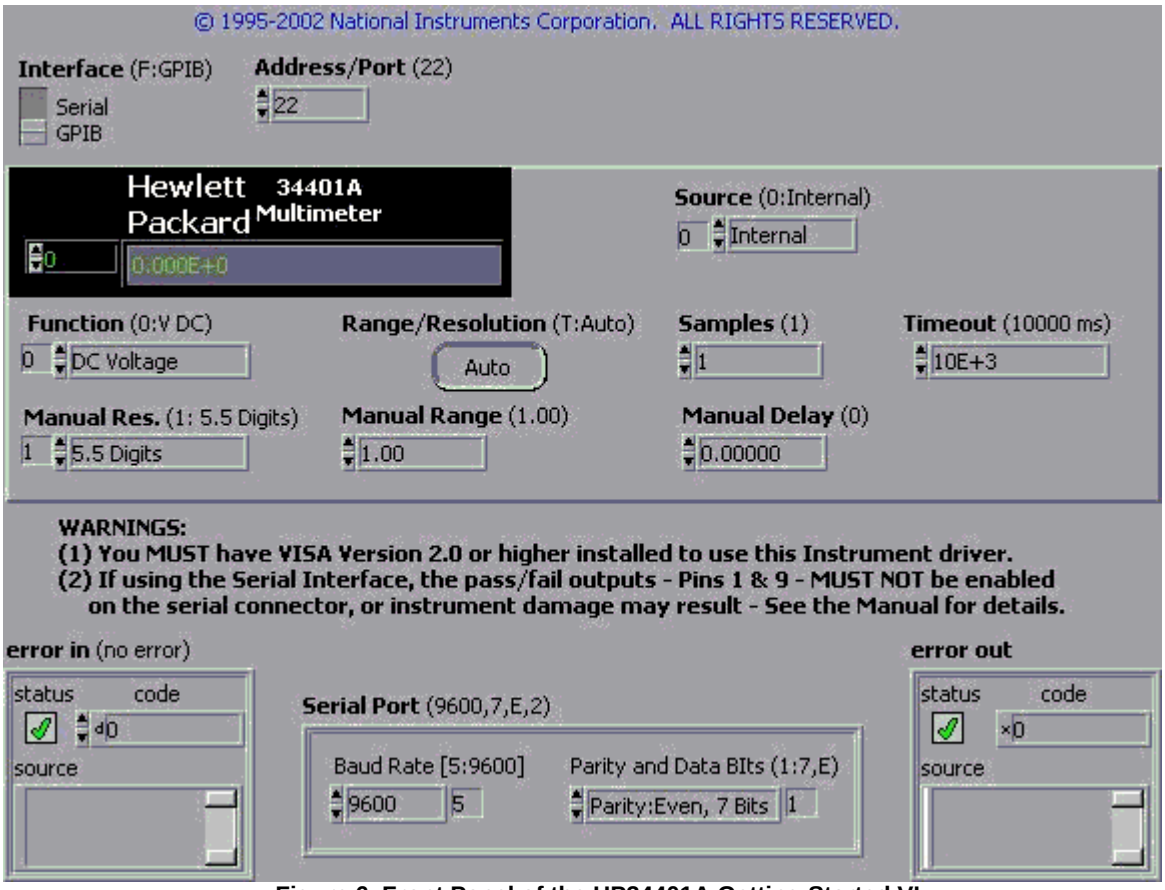

Figure 6. Front Panel of the HP34401A Getting Started VI

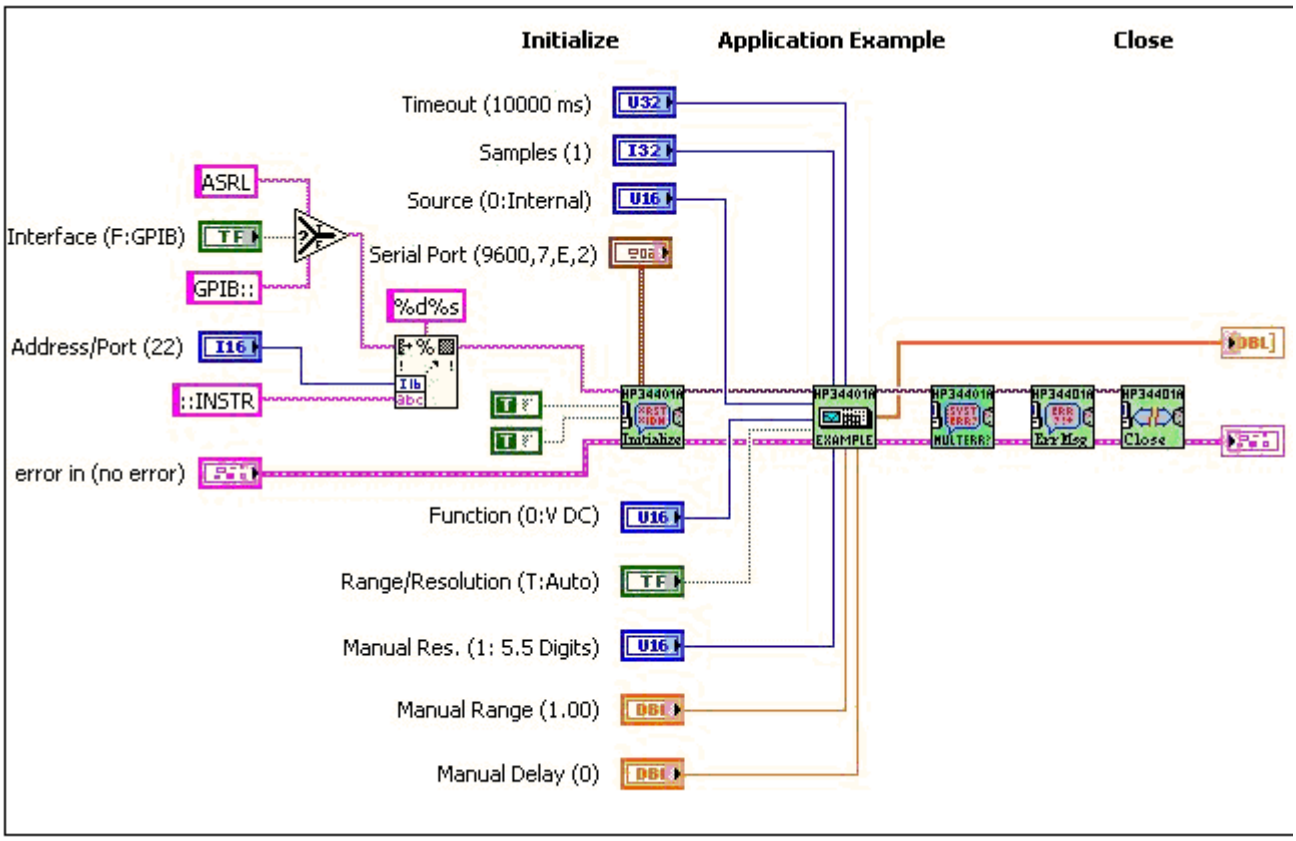

Figure 7. Block Diagram of the HP34401A Getting Started VI

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. Figure 8 shows an example of a VI tree VI.

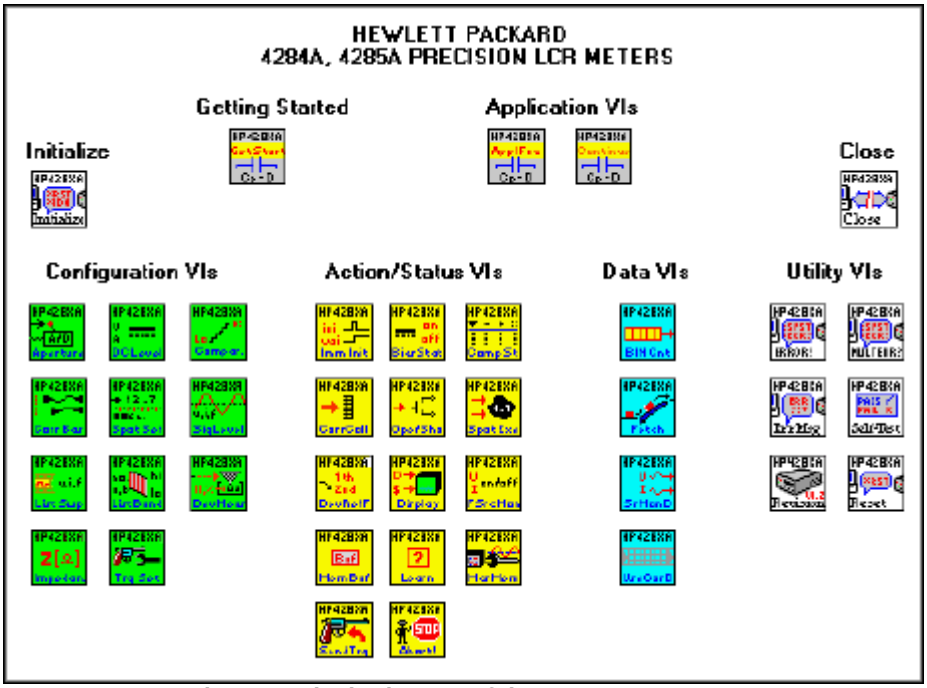

Figure 8. Block Diagram of the HP428xA VI Tree VI

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 section 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 complete the following three steps.

Step 1. Design 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 applications. 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. Refer to the *Instrument Driver Template VIs* section of this document for more information about template VIs.

Develop the 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 might explain how to set up a simple test.

3. Study the programming section of the manual. Read 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 the manual often correspond to the functional groupings of an instrument driver.

Develop a Hierarchy for the Instrument Driver VIs
After you develop the 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.

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 necessary. If you are developing a driver for a non-IEEE 488.2-compliant or a register-based device, you must develop equivalent VIs for your instrument. Refer to your instrument manual for more information about commands and SCPI compliance.

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 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 runs. 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 might not give the user enough control of the instrument operation. Conversely, an instrument driver with many low-level VIs is complicated 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 might 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 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, 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 that is similar to other instruments of the same type both in appearance and functional structure.

  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, 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 instrument manuals. In particular, the programming section of the manual might 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 contains individual VIs for each of the control groups.

A modular driver also contains 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. Modify the Instrument Driver Templates
After you design the LabVIEW instrument driver structure, you must 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.

Complete the following steps to modify the LabVIEW instrument driver template.

1. Download the `CoreDrv.llb` (linked below) containing the latest instrument driver template VIs to the `labview\examples\instr` directory.

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 control and indicator descriptions. Refer to the *LabVIEW Help* for more information about creating descriptions and tip strips for objects. You can access the *LabVIEW Help* by selecting **Help»Contents and Index** (6.0 or earlier) or **Help»VI, Function, & How-To Help** (6.1 or later).

5. Edit the icon. Create an icon for each of the color modes of the icon -- black and white, 16-color, and 256-color. Refer to the *LabVIEW Help* for more information about creating icons.

6. Delete the **Modification Instructions** string control after you have complete 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 of this document for information about error and warning codes to be returned by the VIs.

After completing these steps, 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`, the downloaded files include another 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, rename and modify them like those in `CoreDrv.llb`. Refer to the *Instrument Driver Template VIs* section for a description of each template VI.

Step 3. Add 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. Refer to the *Details for Building Your Instrument Driver VIs* section for design and style details.

Complete the following steps to add your new VIs.

Note: The front panels of the template VIs also include these steps.

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 front panel. Create the controls and indicators for the VI.

3. Edit all control and indicator descriptions. Refer to the *LabVIEW Help* for more information about creating descriptions and tip strips for objects.

4. Edit the icon. Create an icon for each of the color modes of the icon -- black and white, 16-color, and 256-color. Refer to the *LabVIEW Help* for more information about creating icons.

5. Edit the connector pane. Select an appropriate connector pane pattern and wire all controls and indicators to the terminals. Refer to the *LabVIEW Help* for more information about assigning connector pane terminals.

6. Edit the block diagram. Program all operations necessary to carry out the functionality of the instrument driver VI. Editing the block diagram is the most challenging step in adding a component VI to the instrument driver. Defining a block diagram structure makes it easier to edit the source code. You can divide this process into the following steps.

a. Place the appropriate I/O routines on the block diagram.

b. Wire the **error in** control terminal to the first I/O VI **error in** terminal. Then wire the **error out** terminal of that VI to the **error in** terminal of the next VI. Continue this process for all the I/O VIs, and wire the **error out** terminal of the last VI to the **error out** indicator terminal.

c. Wire the **VISA session** terminal to every I/O VI, in a manner similar to step b with the **error in** and **error out** terminals.

d. Use the built-in String functions to assemble a command string based on the VI inputs.

e. Wire the command string to the VISA Write function.

f. Use the VISA Read function to read the response if the instrument generates a response.

g. Use the String functions to parse the response and wire it to the appropriate indicator terminals.

7. Save the VI.

8. Test the instrument driver VI.

9. Repeat steps 1 through 8 for every instrument driver component VI and application VI that you define for your instrument.

10. Edit the instrument driver `.llb` by selecting **Tools»VI Library Manager**. In the VI Library Manager, edit the names of the functions if necessary.

11. Edit the arrangement of icons on the **Functions** palette by clicking the **Options** button on the palette toolbar. Refer to the *LabVIEW Help* for more information about editing palette views.

See Also:
Download instrument driver templates from NI FTP

Details for Building Your Instrument Driver VIs

This section describes layout and style requirements for the three components of an instrument driver VI -- the front panel, the block diagram, and the icon and connector pane.

Front Panel
After 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 have four types of control styles -- Boolean, digital numeric, text ring numeric, or string. For example, any instrument command that has two options (such as `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 control. You need only these four control types 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 each type of control.

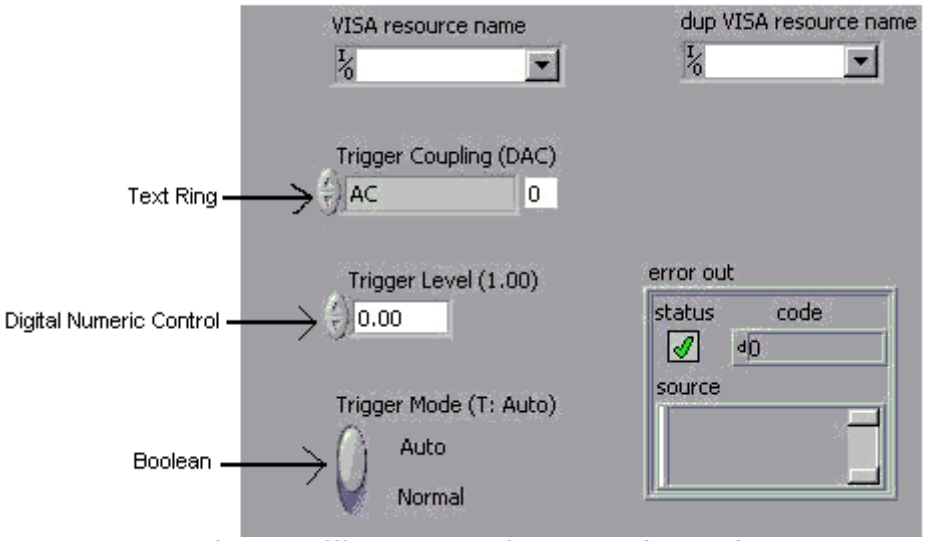

Figure 9. Different Types of Front Panel Controls

In addition to the controls required to operate the instrument, the front panel also must have the following required controls -- **VISA session**, **dup VISA session**, **error in**, and **error out**. Refer to the *Drivers Support Libraries* section for more information about the VISA session handles. Refer to the *Error Clusters* section of Chapter 6, *Running and Debugging VIs*, in the *LabVIEW User Manual* for more information about the **error in** and **error out** parameters. You can access the *LabVIEW User Manual* by selecting **Help»View Printed Manuals** (6.0 or earlier) or **Help»Search the LabVIEW Bookshelf** (6.1 or later).

When designing front panels, use the following style guidelines to ensure uniformity with other LabVIEW front panels. Refer to Chapter 6, *LabVIEW Style Guide*, of the *LabVIEW Development Guidelines* manual for more information about style guidelines.

- Use the default font (application) for all labels because this font is included with LabVIEW and is available to all other users.
- 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, capitalize only the first letter of each word in the control labels. The only exceptions are **dup VISA session**, **error in**, **error out**, and acronyms such as ID or GPIB.
- Enclose default information in parentheses in the control label so the defaults appear in the **Context Help** window, which aids in wiring to the VI on the block diagram. For example, label a function selector ring control whose default is DC volts at item zero **Function (0:DCV)**, and label a Boolean switch that defaults to TRUE indicating automatic **Trigger Mode** (T:Auto). The default information should be 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. Because the error in control is not designed to be used as an interactive input, place it on the left side off screen so it is not visible on the front panel.
- Edit all control and indicator descriptions. Refer to the *LabVIEW Help* for more information about creating descriptions and tip strips for objects.

Block Diagram

After designing the front panel, you must create the code that performs the functionality of the VI. Each type of front panel control has a corresponding function that simplifies the task of building command strings. Rather than wiring a Boolean control to the Select function and selecting a string constant to pass to a Concatenate Strings function, use the Append True/False String function. This function both selects the proper string and concatenates it to the command string in one step. Likewise, use the Format Value function to format and concatenate simple numeric values rather than using one of the To Decimal or To Exponential type functions with the Concatenate Strings function. Again, the Format Value function combines the functionality of the separate conversion and concatenate functions and simplifies the block diagram. For text

rings, use the Append True/False String function, and for string inputs use the Concatenate Strings function. Refer to the *LabVIEW Help* for descriptions of each of these functions. The block diagram in Figure 10 shows the preferred methods of building command strings.

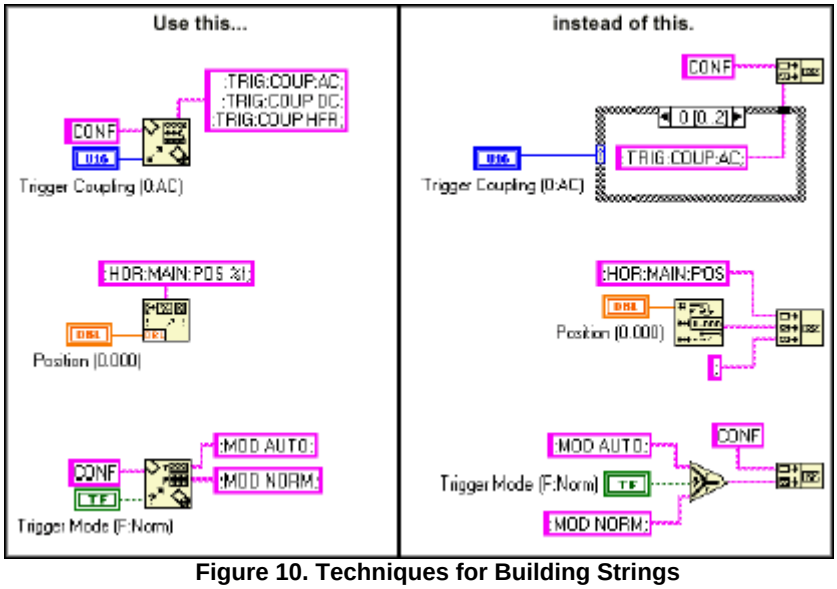

Figure 10. Techniques for Building Strings

Carefully consider the control flow while building the block diagram. Block diagram nodes not connected by wires can execute in any order. Although the *LabVIEW Development Guidelines* manual recommends using a left-to-right and top-to-bottom layout, nodes do not necessarily execute in left-to-right, top-to-bottom order. 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**and **error out**clusters to chain I/O functions together, thus defining the execution order without using Case or Sequence structures, as shown in Figure 4. Although Sequence structures also force the flow of execution, you should avoid them because they hide parts of the block diagram, which makes it difficult for users to understand and modify the block diagram.

Even with proper execution order defined, you might 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 timeout can occur, or the instrument can malfunction. You can solve these internal timing problems in the following 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.

- 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 execution.

- Insert appropriate time delays for instruments that cannot generate interrupts indicating that they are ready for new commands. Because most instruments have input buffers, you usually can send a string containing several commands to the instrument. The individual commands are processed by the instrument serially, or one at a time. Occasionally, an instrument requires a few seconds to finish executing the commands in its buffer before it is ready to accept new commands or respond to a query. Use only the Wait (ms) function to impose a time delay when you cannot configure the instrument to generate a service request when ready and status information is unavailable.

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

When building block diagrams, use the following style guidelines to improve the appearance and ensure ease of understanding. Refer to Chapter 6, *LabVIEW Style Guide*, of the *LabVIEW Development Guidelines* manual for more information about style guidelines.

- Use proper wiring style. Do not crowd the block 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 type terminals when possible. You can use the arrow keys to move objects by single pixels if necessary. Use

the **Align Objects** and **Distribute Objects** buttons in the toolbar to add symmetry and straight lines to the block diagram.

- Add text labels to each frame of Case and Sequence structures. Alternatively, you can use enumerated type controls, which provide meaningful descriptions to cases within a Case structure.
- Label long wires and complex operations as necessary to increase clarity.
- Label control and indicator terminals with normal text, but use **bold** text to make free label comments stand out. The background of labels should be colored transparent. Figure 11 is an example of a block diagram that uses these style guidelines.

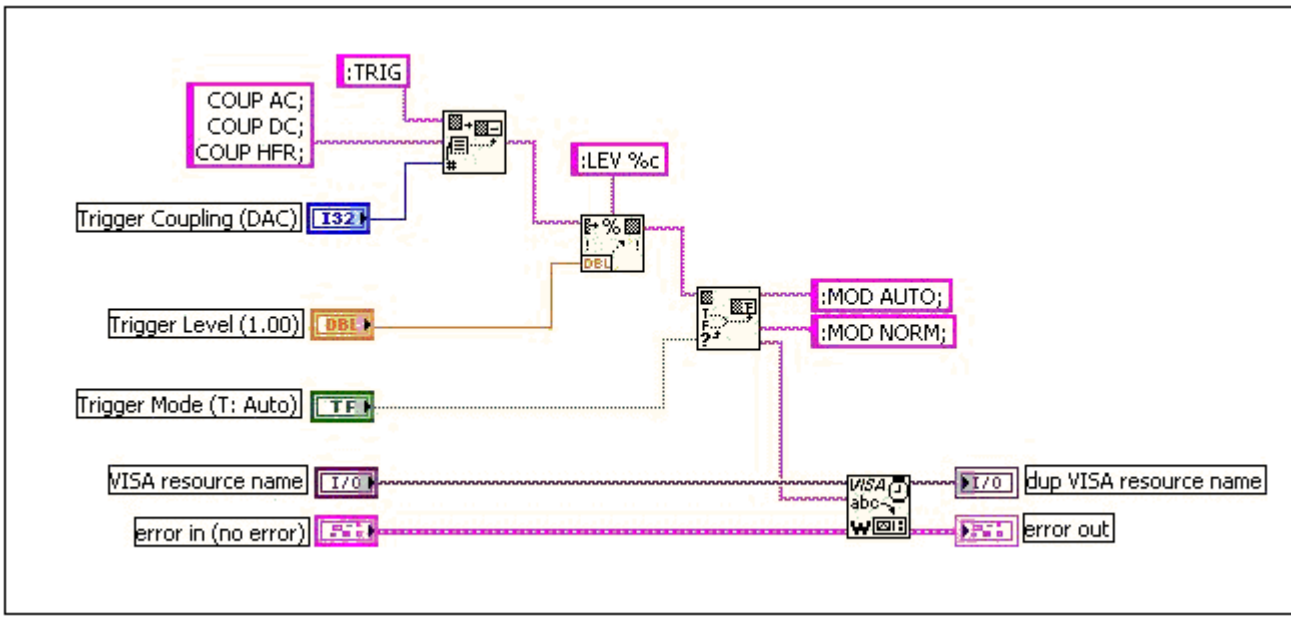

Figure 11. Simple Trigger Block Diagram Following Style Guidelines

Icon and Connector Pane

Reserve the upper left terminal of the connector pane for the **VISA Session** control and the upper right terminal for the **dup VISA Session** indicator. Reserve the lower left terminal for the **error in** control and the lower right terminal for the **error out** indicator to simplify wiring to subsequent error terminals. Select a connector pane pattern that has more terminals than the number of controls and indicators because you might need to add controls or indicators to the connector pane at a later time. 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 to promote a left-to-right data flow on 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 `insticon.llb` located in the `labview\examples\instr` directory. Be sure to include text in the icon containing the instrument model controlled by the VI. If you cannot create an icon to represent the functionality of the VI, you can use text. Figure 12 shows examples of icons.

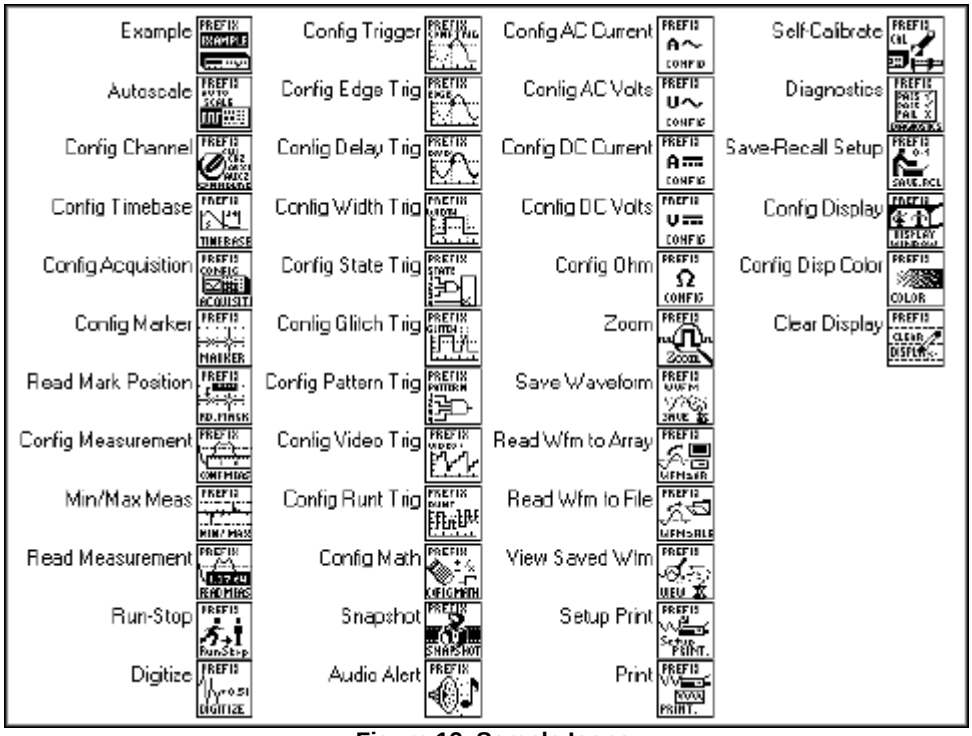

Figure 12. Sample Icons

See Also:

Product Manuals: LabVIEW User Manual
Product Manuals: LabVIEW Development Guidelines

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. Avoid attempting to make 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; do not reproduce their code in the block diagram of the application VI. Also, do not use the instrument driver initialize or close VIs within the application VI because this makes it less useful in higher level applications.

Documenting VIs
To aid the user, you must include **Context Help**window information for each instrument driver. Refer to the *LabVIEW Help* for information about documenting VIs.

Users access the **Context Help**window by selecting **Help»Show Context Help**When you move the cursor over front panel or block diagram objects or the icon in the upper right corner of the window, a description of that object appears in the **Context Help**window, provided that you configuring a description for the object.

You also can help the user by placing free labels on the front panel and the block diagram. On the block diagram especially, you should show all terminal labels (plain text) and color the borders transparent. Place free labels in Case and Sequence structures using bold text. This makes the comment stand out and makes the VI easier to understand and modify. Also, enumerated type controls provide additional self documentation on the block diagram.

Error Reporting
The VISA functions check 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 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 passes to the **error out**cluster; otherwise, the **error in**information passes out of the function. 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) pass through without triggering error actions. Refer to the *LabVIEW Help* for a list of VISA error codes.

In addition to VISA error codes, the error and warning codes in Table 3 are reserved for instrument drivers. 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. To be more VXIPnP-compliant, use the new codes.

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
3FFC0800to 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
BFFC0804to BFFC0FFF	-1073999873 to -1074001916	ERROR: Instrument specific errors	Instrument Driver

Before 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, ease of use, 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. Most instrument drivers developed today use the query method to report errors.

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 can 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. Querying the instrument for errors adds to the execution time of the VI, but this technique is beneficial for detecting instrument-specific errors. The only disadvantage to this method is that 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 because it flushes the instrument 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 generate an appropriate error message. Figure 13 shows the front panel of the Error Query (Multiple) VI.

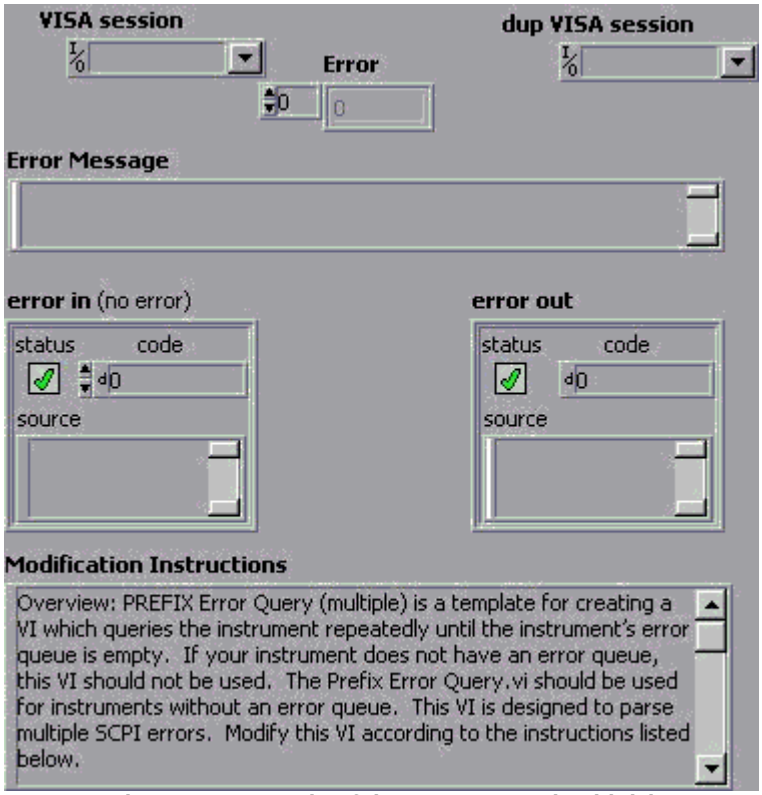

Figure 13. Example of the Error Query (multiple) VI

To use the VI in your application, place it wherever you want to query the instrument for errors. During initial development, you might 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 execute. Because SCPI instruments buffer the errors in a queue, the error query VI can 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 the block diagrams; use the capabilities of the instrument for this. However, if 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 you can use to meet that goal.

While error querying is a very effective method for detecting and reporting errors, it has some limitations. Not all instruments have SCPI-defined error queues. For these instruments, you must modify (or replace) the Error Query VI with one of your own design to accommodate the capabilities of your instrument. Also, some of the instrument messages might not be specific enough to be of any practical value. For example, the instrument might report only a generic parameter error when it detects a value out of range; this is not especially helpful if a VI has 10 numeric controls on it. Finally, you must be careful about using the information in an instrument error queue. You must make sure the information is current and that it is not 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, which might be read at a later time and misinterpreted as occurring after when it actually happened.

Additional Style Guidelines

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 might need to modify the code to optimize it for their special needs. Consider using the following guidelines to aid the user. Refer to Chapter 6, *LabVIEW Style Guide*, of the *LabVIEW Development Guidelines* manual for more information about style guidelines.

- Except for **error in** and **error out**, avoid using cluster controls and indicators. 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 connector pane, you still should try to avoid using clusters. You should either re-evaluate the grouping of the inputs for the VI or use some terminals on the right side of the connector

pane. Use clusters only when there is a logical grouping of controls, such as the error cluster, which you will pass and use in several VIs.

- Use color sparingly. Although your development computer might have millions of colors, 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 computer might have a high resolution monitor, the application machine might have only a resolution of 640x480. During development, make sure front panels and block diagrams are readable and fit on various platforms.

- Use graphics sparingly on front panels. These VIs later will be used as subVIs in a final application. The user generally will modify the front panels to create his own front panels to be seen by the application operators. Graphics make it difficult for the user to easily modify the front panel.

- Set the tabbing 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.

- Do not crowd the block diagram. Crowded diagrams and front panels are more difficult to understand than a simple and neatly organized VI. Also allow extra space around items with labels in order to account for font sizing differences with different printers or systems. Set all labels to **Size to Text**

Creating Palette Views
To make it easier for customers to install, access, and use instrument drivers, create palette views for your instrument driver. For consistency, instrument drivers should appear in the **Instrument Driver VIs**ubpalette. Within the subpalette, the instrument VIs should have the same organization as the internal design model as shown in Figure 14. 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 14. Example Subpalette for the HP34401A

To achieve the same palette structure for all instrument drivers, start with the template menu files. Place the template menu files and your instrument driver files in a new directory in the `labview\instr.lib` directory. Restart LabVIEW to see the template VIs in the **Instrument Driver VIs**ubpalette. For each subpalette, insert the VIs that correspond to that category. For instrument drivers with many subVIs, 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 or copy the VIs from the temporary palette onto each component group subpalette.

Testing the Operation
You should test your instrument driver as you develop it. Although most users will use the **Context Help**window to determine the inputs to the VIs, some might be confused and pass invalid data to the VI. Therefore, you also should test your VIs with invalid data, boundary conditions and ranges, and unusual combinations of inputs. Similarly, if a subVI needs string or array information, test an empty array or empty string.

See Also:
Product Manuals: LabVIEW Development Guidelines

Driver Support Libraries

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 and functions. The main items of interest are the VISA functions and the Instrument Driver Template VIs.

VISA
The VISA functions 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, TCP/IP, and other types of instruments. Refer to the *LabVIEW Help* for descriptions of VISA functions and controls.

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 subVIs. The VISA session identifies the resource being operated on by the VI. It also differentiates between different sessions of the instrument driver.

- **VISA Session** (except for the initialize VI) 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.

- **dup VISA session** 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.

You pass the VISA session parameters into and out of VISA functions on the block diagram.

Instrument Driver Template VIs
The LabVIEW instrument driver templates, located in the `CoreDrv.llb` and `CoreDrU.llb` that you downloaded the `labview\examples\instr` directory, contain a set of VIs common to most instruments. Because these templates are updated periodically, you should download the latest version from National Instruments FTP site (linked below). 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 instructions for modifying 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. You can use the instrument drivers 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.

Refer to the **Context Help** window for information about each template VI.

See Also:
Download instrument driver templates from NI FTP

Conclusion

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 you actually would use the instrument. Ideally, with this type of structure, users have on each individual front panel exactly what they need to perform the particular instrument operation. The greatest challenge in developing instrument drivers lies in determining which controls belong on each particular VI.

For the user, the logical structure, documentation, 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 block 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 several 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 VIs. These proven tools help you design instrument drivers that are acceptable to a wide range of users.

Top