Instrument I/O VIs

Click here for Instrument I/O Overview Topics.

Instrument Driver Template VI Descriptions
VISA Library Reference Function Descriptions
GPIB Function Descriptions
GPIB 488.2 Function Descriptions
Serial Port VI Descriptions

Instrument I/O Overview Topics

LabVIEW Instrument Drivers Overview
Developing a LabVIEW Instrument Driver
Instrument Driver Template VI Overview
VISA Overview
GPIB Overview
GPIB 488.2 Overview
Serial Port VI Overview

VISA Library Reference Functions

VISA Library Reference Function Descriptions

GPIB Functions

Traditional GPIB Function Descriptions

GPIB 488.2 Functions

GPIB 488.2 Function Descriptions

Serial I/O VIs

Serial Port VI Descriptions

LabVIEW Instrument Drivers Overview

This topic contains an overview of the LabVIEW instrument driver model and VIs.

Introduction to Instrument Drivers
LabVIEW Instrument Driver External Interface Model
LabVIEW Instrument Driver Internal Design Model
Instrument Driver Distribution

Introduction to Instrument Drivers

A LabVIEW instrument driver is a set of VIs that control a programmable instrument. Each VI corresponds to a programmatic operation such as configuring, reading from, writing to, or triggering the instrument. LabVIEW instrument drivers simplify instrument control and reduce test program development time by eliminating the need to learn the low-level programming protocol for each instrument.

Two conceptual models help define a standard for LabVIEW instrument driver software design, development and use. The first model, the instrument driver external interface model, shows how the instrument driver interfaces with other system components. The second model, the instrument driver internal design model, defines the internal organization of an instrument driver software module.

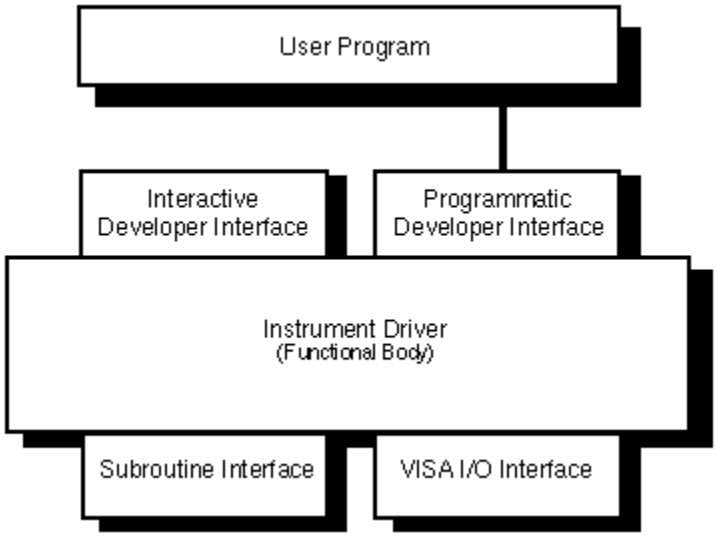
The LabVIEW instrument driver library contains instrument drivers for a variety of programmable instrumentation, including GPIB, VXI, and serial. If a driver for your instrument is in the library, you can use it as is to control your instrument. Instrument drivers are distributed with their block diagram source code, so you can customize them for your specific application. If a driver for your particular instrument does not exist, you can:

- Try using a driver for a similar instrument. Often similar instruments from the same manufacturer have similar if not identical instrument drivers.
- Modify the Instrument Driver Template VIs to create a new driver for your instrument.
- Use either the GPIB, VXI, Serial, or VISA I/O libraries provided with LabVIEW to send commands directly to your instrument.

LabVIEW Instrument Driver External Interface Model
LabVIEW Instrument Driver Internal Design Model
Instrument Driver Distribution

LabVIEW Instrument Driver External Interface Model

The following figure shows a general model of how a LabVIEW instrument driver interfaces to the rest of the system:



Functional Body
Interactive Developer Interface
Programmatic Developer Interface
I/O Interface
Subroutine Interface

Functional Body

The *functional body* is the actual code for the instrument driver. Refer to the LabVIEW Instrument Driver Internal Design Model topic for more information.

The most successful instrument driver products have, historically, been developed by using a standard programming language for the functional body. This is the approach LabVIEW instrument drivers use. The advantages include greater developer control over the driver, more robust drivers, and increased functionality. LabVIEW instrument drivers are written using the standard LabVIEW graphical programming environment.

The functional body of a LabVIEW instrument driver is a set of VIs that control a specific instrument. The source code for these VIs are block diagrams consisting of executable icons connected by data flow wires. Because the functional body is developed with the standard tools provided in LabVIEW, users can easily view instrument driver source code and optimize it for their application.

Interactive Developer Interface

The interactive developer interface of a LabVIEW instrument driver is the front panel. It is analogous to a physical instrument panel and is the interactive user interface of the VI. On the front panel, controls and indicators graphically represent the inputs and outputs of the VI. With the LabVIEW front panel, users can interactively operate individual instrument driver VIs and verify communication.

Programmatic Developer Interface

The icon/connector is the programmatic interface of the LabVIEW instrument driver VI. It consists of a graphical representation of the VI (icon) and a definition of the input and output terminals for the VI (connector). When you call or execute a VI from another VI, you place a copy of the subVI icon/connector in the block diagram of the calling VI. Information passes between the two VIs through the connector terminals. There are several benefits to this approach. You can easily assemble test systems utilizing LabVIEW instrument drivers by combining a few instrument driver VIs, each using multiple parameters. The instrument driver interface in the user program is modular and easy to identify, and you can recall the

VI front panels during debugging to understand how the program uses the instrument driver.

I/O Interface

An important consideration for instrument drivers is how they perform their I/O to and from instruments. The I/O interfaces for LabVIEW instrument drivers are the VISA and GPIB function libraries, and the VXI and Serial VI libraries. These libraries contain sets of functions and VIs that cover the capabilities of GPIB, VXIbus, and Serial bus capabilities, including both message-based and register-based programming, interrupt and event handling, and direct access to the VXI backplane.

VISA, an acronym for Virtual Interface Software Architecture, is a single interface library for controlling VXI, GPIB, RS-232, Ethernet, and other types of instruments.

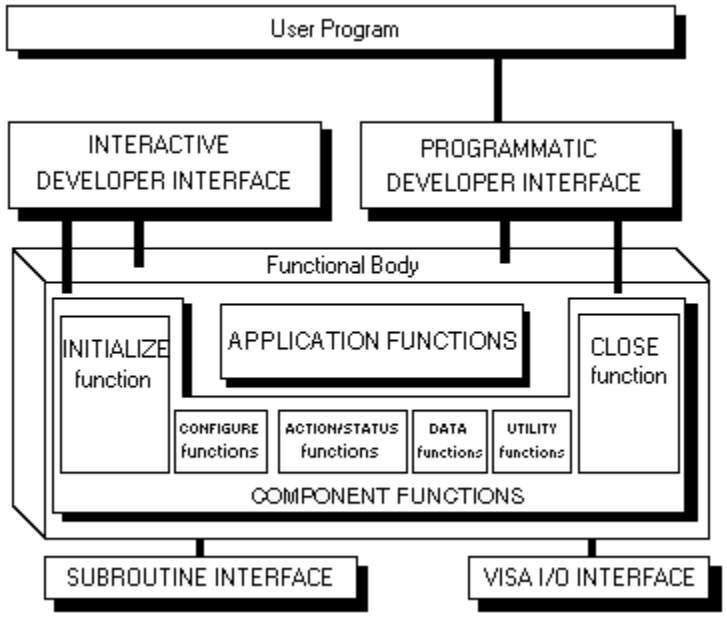
Subroutine Interface

Because you write LabVIEW instrument drivers in standard LabVIEW graphical code, an instrument driver is a software program with the same capabilities as any other LabVIEW VI. While some VIs (such as instrument drivers) perform only simple I/O to and from an instrument, other VIs may control multiple instruments or use support libraries to integrate data analysis or other measurement-specific operations. With LabVIEW, you can build virtual instruments that combine hardware and software capabilities. You can develop and package complete, high-level tests as single VIs, which other test developers can reuse.

By ensuring compatibility with the virtual instrument concept, the LabVIEW instrument driver standard has unlimited potential for delivering baseline as well as sophisticated application-specific instrument drivers. The LabVIEW instrument driver standard defined in this document applies both to instrument drivers that control only a single instrument, and to virtual instrument drivers that combine instrument drivers:external interface model:subroutine interfaceinstrument drivers:external interface model:subroutine interface features of multiple instruments and add software processing.

LabVIEW Instrument Driver Internal Design Model

The LabVIEW instrument driver internal design model, shown in the following figure, defines the organization of the LabVIEW instrument driver functional body. Because development guidelines and all LabVIEW instrument drivers are based on this model, it is important to both developers and end users of instrument drivers. When you understand the model and how to use one instrument driver, you can use that knowledge across numerous instrument drivers.



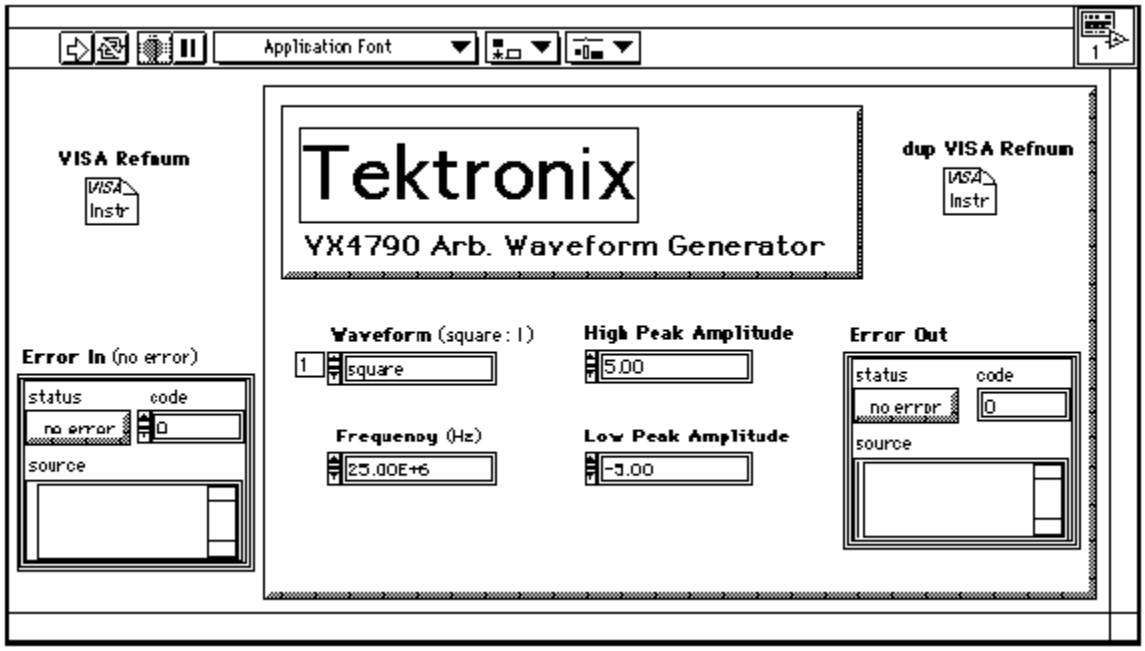
The functional body of a LabVIEW instrument driver consists of two main categories of VIs. The first category is a collection of component VIs, which are individual software modules that each control a specific type of instrument function. The second category is a collection of higher-level application VIs that show how to combine the 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 instrument drivers:internal design model:functional bodyinstrument drivers:internal design model:functional bodyseveral times.functional body of instrument drivers:internal interface modelfunctional body of instrument drivers:internal interface model

Instrument Driver Application VIs
Instrument Driver Component VIs
Error Reporting

Instrument Driver Application VIs

The *application VIs* are at the highest level of the instrument driver hierarchy. They are written in LabVIEW block diagram source code and control the most commonly used instrument configurations and measurements. These VIs serve as a code example for how to configure the instrument for a common operation, trigger the instrument, and take measurements. Because the application VIs are standard VIs, with icons and connector panes, you can call them from any higher-level application when you want a single, measurement-oriented, interface to the driver. For many developers, the application VIs are the only instrument driver VIs needed for instrument control. The Tek VX4790 Example VI, shown in the following figure, demonstrates an application VI front panel.



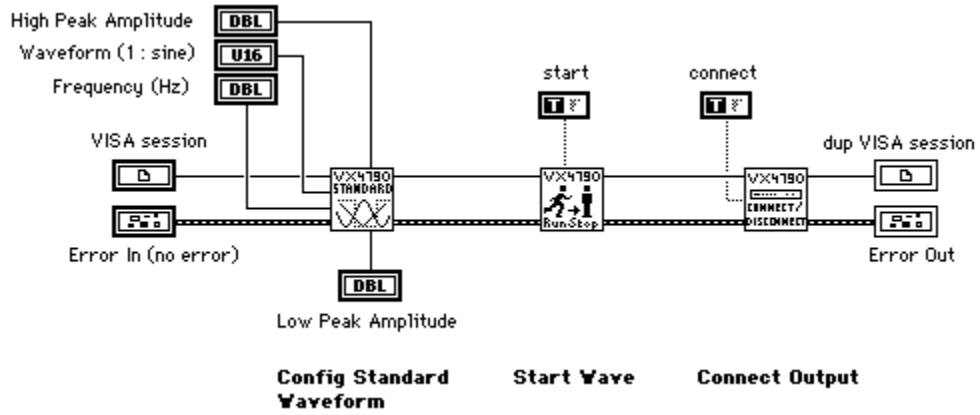
The application VIs are built from a lower-level set of instrument driver *component VIs*.instrument drivers:internal design model:application VIsinstrument drivers:internal design model:application VIs

Instrument Driver Component VIs

LabVIEW instrument drivers have component VIs, which are a modular set of VIs that contain all of the instrument configuration and measurement capabilities. The component VIs fit into six categories: initialize, configuration, action/status, data, utility, and close.

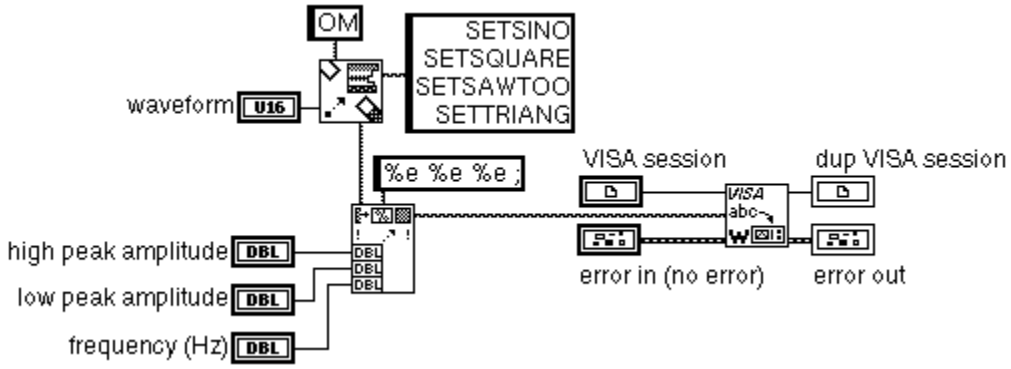
Each of these categories, with the exception of the initialize and close VIs, consists of several modular VIs. Most of the critical work in developing an instrument driver lies in the initial design and organization of the instrument driver component VIs. The specific routines in each category are further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

The following figure shows how the Tek VX4790 Example application VI diagram uses the instrument driver component VIs:



The block diagram of the instrument driver component VIs uses standard LabVIEW VIs, as well as VISA VIs to build command strings and send them to the instrument. In the following figure, the Tek VX4790 Config Std Wave component VI block diagram assembles the command string and wires it into the VISA Write function. This function performs the necessary I/O, checks for errors, and updates the appropriate

error indicators.



Error Reporting

LabVIEW instrument drivers use error clusters to report all errors, as shown in the following figure. 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 all the way through your instrument driver and out to your full application.

Another benefit of error input/output is that data dependency is added to VIs that are not otherwise data dependent.

Instrument Driver Distribution

LabVIEW instrument drivers are distributed in a variety of media including electronic via bulletin board and internet and CD-ROM.

Instrument Driver Bulletin Board Access

You can download the latest versions of the LabVIEW instrument drivers from one of the National Instruments bulletin boards. The bulletin boards are located in National Instruments offices in the United States, United Kingdom, and France. All of the bulletin boards can auto baud to 9,600 baud and use 8 data bits, 1 stop bit, and no parity. The United States bulletin board handles up to 14,400 baud. Telephone numbers for the bulletin boards are:

United States	(512) 794-5422
United Kingdom	(0635) 551422
Outside the United Kingdom	(44) 635 551422
France	(1) 48 65 15 59
Outside France	(33) 1 48 65 15 59

Instrument Driver Internet Access

If you have internet access, you can download the latest instrument driver files from the National Instrument File Transfer Protocol (FTP) site. Use FTP to log on to host `ftp.natinst.com`. Use anonymous as the user name and your e-mail address as the password. The instrument drivers are in the

following locations:

LabVIEW for Windows
`/support/labview/windows/instruments/`

LabVIEW for Macintosh
`/support/labview/mac/instruments/`

LabVIEW for Sun
`/support/labview/sun/instruments/`

CD-ROM Instrument Driver Distribution

The entire library of LabVIEW instrument drivers is available on CD-ROM. The instrument driver CD-ROM is available from National Instruments at no charge. You can retrieve the latest instrument driver list on a touch-tone phone by calling the National Instruments automated fax system, Fax Back, at (512) 418-1111 or by calling National Instruments.

Initialize VI

All LabVIEW instrument drivers should have an initialize VI. It is the first instrument driver VI called, and establishes communication with the instrument. Additionally, it can perform any necessary actions to place the instrument either in its default power on state or in some other specific state.

Configuration VI

The *configuration VIs* are a collection of software routines that configure the instrument to perform the desired operation. There may be numerous configuration VIs, depending on the particular 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 trigger system or generating a stimulus. These VIs are different from the configuration VIs because they do not change the instrument settings, but only 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. The specific routines in this category and the actual operations they perform are left up to you.

Data VIs

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 and the actual operations performed by those routines are left up to you.

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 instrument driver template VIs such as reset, self-test, revision, error query, and error message and may include other custom instrument driver VIs, such as calibration or storing and recalling setups.

Close VI

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

Template VIs

The *template VIs* are instrument driver VIs that you can use as templates or examples. 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 Instrument Driver Template VIs topic.

Developer-Specified VIs

The remainder of instrument driver VIs are known as developer-specified VIs, and the actual operations performed by those routines are left up you. 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.

Developing a LabVIEW Instrument Driver

This topic describes the procedure for developing a LabVIEW instrument driver. The ideal LabVIEW instrument driver has full function control of the instrument. Rather than mandate the required functionality of all instrument types, such as DMMs, counter/timers, and so on, the following topics focus 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.

Development Procedure
Tips for Developing a LabVIEW Instrument Driver
LabVIEW Instrument Driver Standards Checklist

Development Procedure

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.

Instrument Driver Structure Design

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 since 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. For more information about template VIs see the Instrument Driver Template VIs topic.

Instrument Driver Structure and VI Hierarchy
Guidelines and Recommendations
Design Example

Instrument Driver Structure and VI Hierarchy

When you develop a LabVIEW instrument driver, it is important to clearly define 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 an in-depth knowledge of the instrument operation and use in test applications. The following steps outline one approach to developing the structure for the 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 set-up 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 instrument drivers for similar instruments. Often instruments from the same family have the same programming command set and you can easily modify their corresponding instrument drivers.
5. Determine which LabVIEW template VIs are suitable for use with your instrument.
6. 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

Instrument Driver VI Organization

After you have developed your Instrument Driver structure, you can develop a VI hierarchy to organize the VIs that will be necessary to create 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 common to all types of instrumentation are performed by the template instrument driver VIs: initialize, close, reset, self-test, revision, error query, and error message.

The template VIs for LabVIEW instrument drivers include prewritten VIs to perform these common instrument operations. The command strings are based on IEEE 488.2 Common Commands. 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, little 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, 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.

VI Hierarchy	**Type**
Initialize VI	(Template)
Application VIs	
Autosetup and Read Waveform	(Developer Defined)
Rise-Time/Fall-Time Measurement	(Developer Defined)
Configuration VIs	
Configure Vertical	(Developer Defined)
Configure Horizontal	(Developer Defined)
Configure Trigger	(Developer Defined

Configure Acquisition Mode	(Developer Defined)
Autosetup	(Developer Defined)
Action VIs	
Acquire Data	(Developer Defined)
Data VIs	
Read Waveform	(Developer Defined)
Voltmeter Measurement	(Developer Defined)
Counter/Timer Measurement	(Developer Defined)
Utilities VIs	
Reset	(Template)
Self-Test	(Template)
Revision	(Template)
Error Query	(Template)
Error Message	(Template)
Close VI	(Template)instrument driver development:design of structure:organization

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. Additional VIs included 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 to both configure the instrument and 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 multiple measurements.
- Concentrate on the correct level of granularity of 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.
- 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 multi-channel 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.instrument driver development:design of structure:guidelines and recommendations

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 instruments 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 a 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.

The following table shows how the command summary from the *Tektronix VX4790 Arbitrary Waveform Generator Operating Manual* relates to developer specified instrument driver VIs.

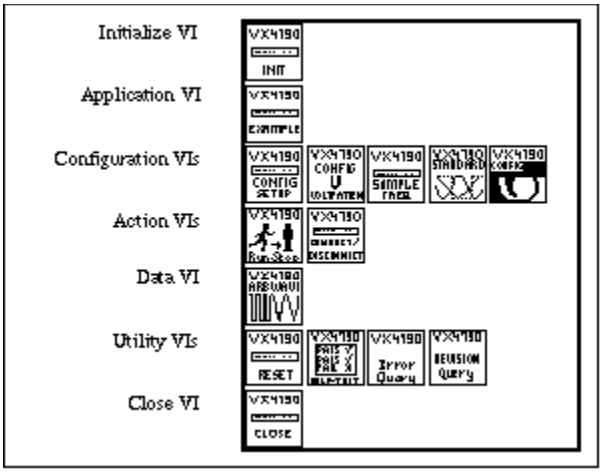
Instrument Manual Section	**Instrument Driver VI**
Setup Commands External clock input enable External trigger source Sync pulse control Isolation relay control	TKVX4790 Setup
Pre-Programmed Waveform Commands Sine wave Square wave Triangle wave Sawtooth wave	TKVX4790 Config Std. Waveform
Frequency Commands Frequency Period Divide	TKVX4790 Config Sample Frequency

Low-Pass filters	
Voltage/Attenuator Commands Voltage control Attenuator enable Attenuation level	TKVX4790 Config Volt/Atten.
Arbitrary Waveform Commands Sample voltage Breakpoint/Last commands	TKVX4790 Download Arb. Waveform
Trigger Commands Start location Breakpoint/last commands	TKVX4790 Run/Stop

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.

The following figure shows the LabVIEW instrument driver VIs for the Tektronix VX4790 Arbitrary Function instrument driver development:design of structure:exampleGenerator:instrument driver development:design of structure



Instrument Driver Template Modification

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 VI in the file `insttmpl.llb`.
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 Instructions to Modify This VI string control on the initialize 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 Instructions to Modify This VI 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, and error message VIs.

After completing this procedure, you will have a base level driver that implements all template instrument driver VIs and is a good framework from which you can create your driver.

Instrument Driver Component VI Additions

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 that you create will be added to the source code along with the template VIs in the file `prefix.llb`.

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 the file insttmpl.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.

Editing the 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 block diagram 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.
4. Use the LabVIEW string VIs to assemble a command string based on the VI inputs.
5. Wire the command string to the Write I/O VI.
6. Use the Read I/O VI to read the response if an instrument response is generated.
7. Use the string VIs to parse the response and wire it to the appropriate indicator terminals.

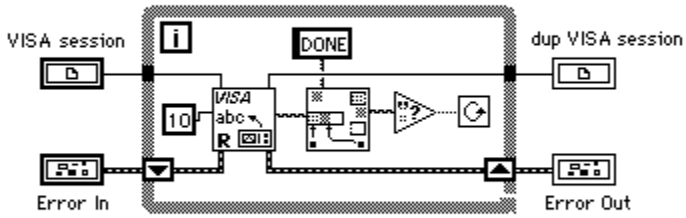
Tips for Developing a LabVIEW Instrument Driver

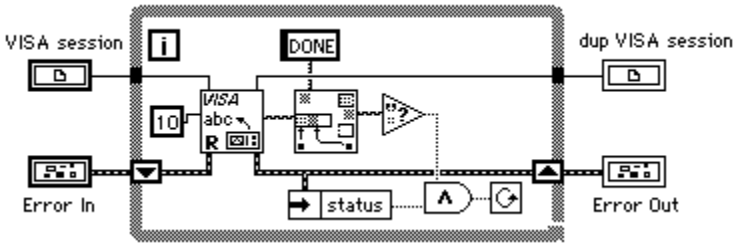
Loop Termination Conditions
Assembling Command Strings
Data Dependency
Programmatically Checking Parameter Ranges
Guidelines
Online Help Information
Application VIs

Loop Termination Conditions

When you use looping structures in instrument driver block diagrams, you must include a way to escape from While Loops if an error occurs. This escape method is important if you are using a While Loop containing I/O routines and the loop termination depends on the result of the I/O.

If there is an error, the I/O routines automatically shut down and LabVIEW may be stuck in an endless loop. Therefore, always test the error cluster status in conjunction with your normal loop termination condition to determine when to terminate the loop. The following two figures show incorrect and correct mechanisms for terminating a Loop, respectively:

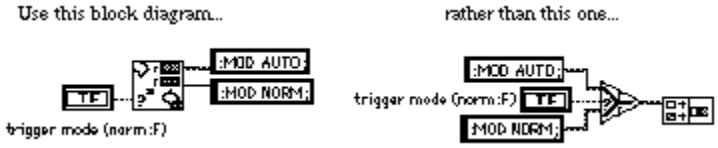




Assembling Command Strings

After you develop your front panel, the next step is to create the block diagram which performs the function required by the VI. Each type of front panel control has a corresponding block diagram string VI that simplifies the task of building command strings.

You can use Pick Line & Append to choose from a selection of strings and concatenate it to another string in a single step. This procedure is easier than using a Case structure and Concatenate Strings.

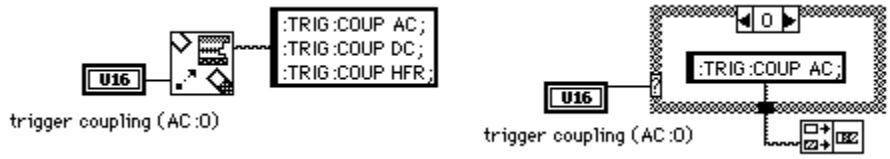


You can use Format & Append to format and concatenate simple numeric values. This procedure is easier than using one of the To Decimal or To Exponential type conversion VIs in conjunction with Concatenate Strings.

Use this block diagram... rather than this one...

By using Select & Append you can select a string constant and concatenate it to another string in a single step. This procedure is easier than using Select and Concatenate Strings.instrument driver development:assembling command strings

Use this block diagram... rather than this one...

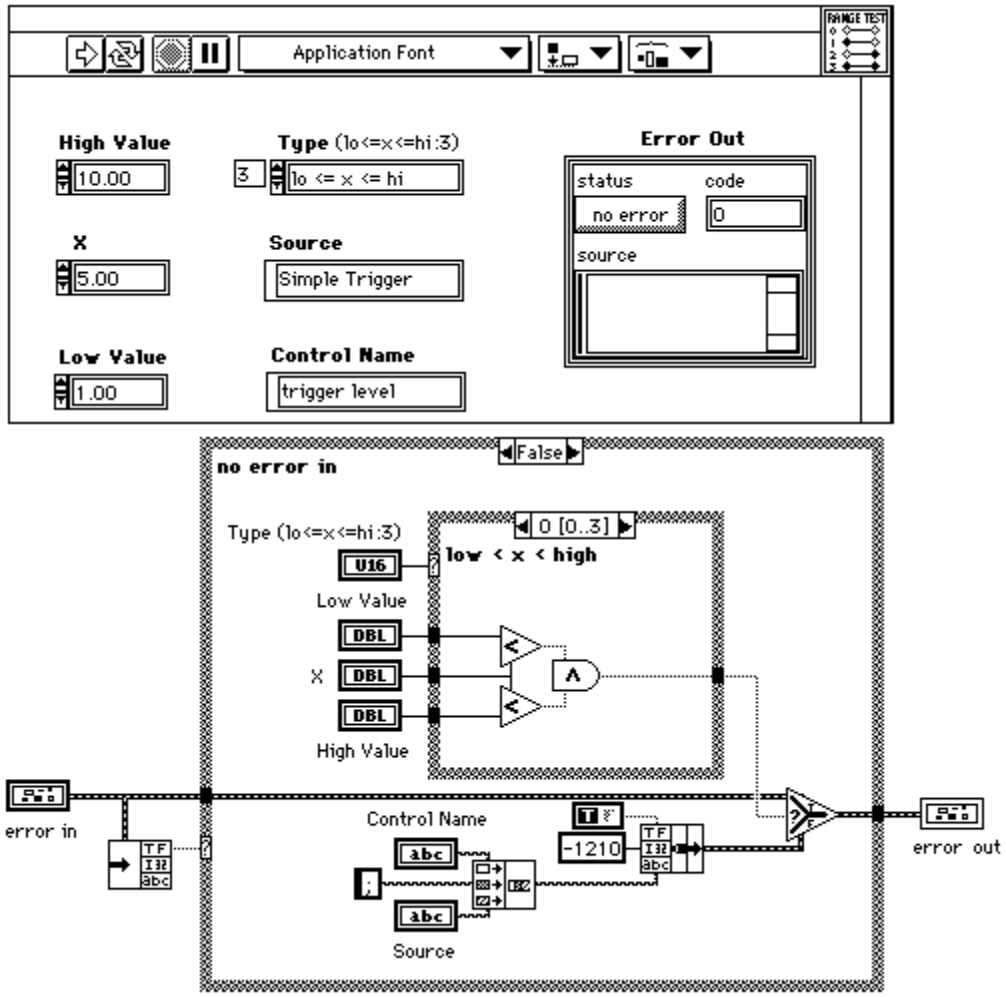


Data Dependency

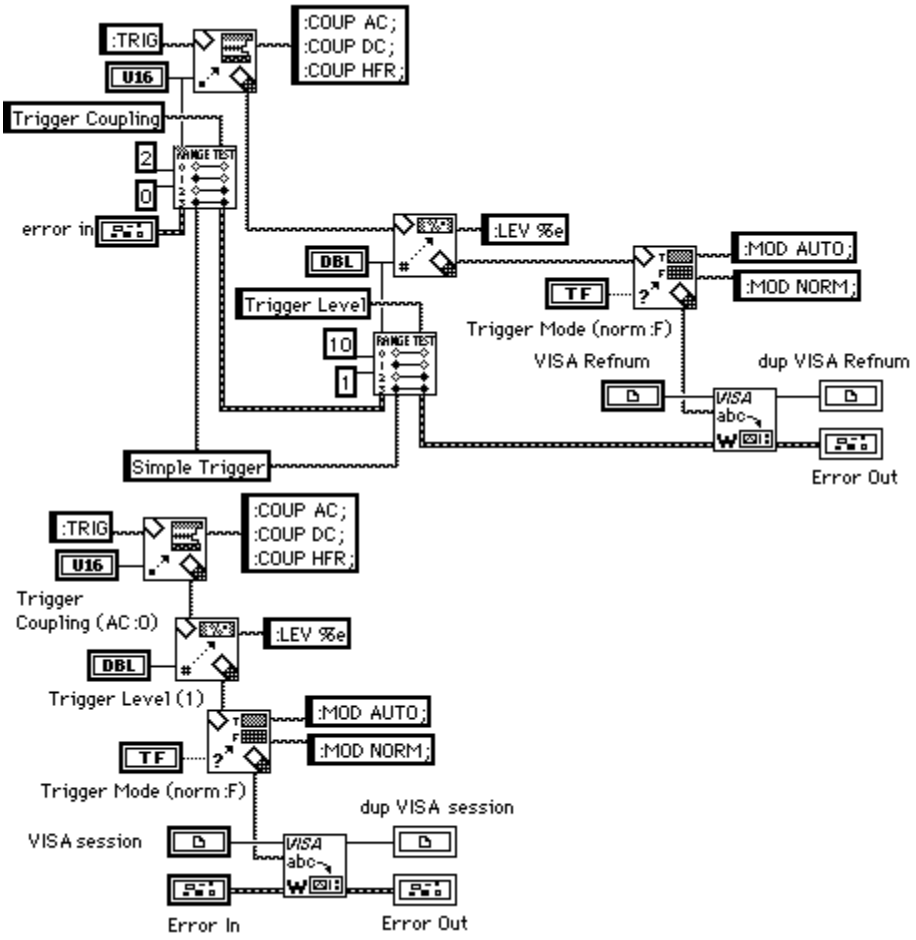
Carefully consider the control flow when you build your diagrams. LabVIEW does not necessarily execute in a left-to-right, top-to-bottom fashion. Data dependency automatically determines execution order. Add artificial data dependency wherever. By using the clusters to chain I/O VIs together, you can define the execution order without using Case or Sequence structures, as illustrated in the figure, VIs in Tek VX4790 Example Diagram. Sequence structures, which hide parts of the diagram, are also effective at controlling execution order. Whichever method you use, make sure that you clearly define control flow so that the correct branch of the diagram executes first.

Programmatically Checking Parameter Ranges

Be prepared to handle unexpected values entered into controls. The front panel **Data Range...** item is a deterrent to many range errors. When circumstances prevent its use, you can either build a block diagram routine based on the Range Test VI shown in the following figures to report range errors to the error cluster, or you can build block diagram routines to coerce the data into range. Also, you must often account for control interaction by using case statements to alter the block diagram routines based on the setting of a particular control. Ideally, you can run the VI with any combination of settings and values without crashing the instrument. If not, add more error detection/correction to your block diagrams. Even though programmatic range checking can simplify the VI use, it can also complicate the development of the VI.



Programmatic range checking can easily double the size of your VI and add some execution speed penalties. The following figures show the changes made to the Simple Trigger VI to programmatically check the ranges of the numeric inputs.



Guidelines

Like the LabVIEW VI, the standard components of an instrument driver VI are the front panel, block diagram, and icon/connector pane.

Front Panel
Required Front Panel Controls
Block Diagram
Icon
Connector Pane

Front Panel

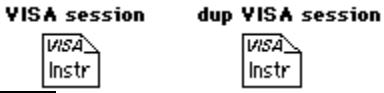
Each VI in your instrument driver should contain a front panel that groups all the necessary controls together to perform the function of the VI. When you develop an instrument driver VI, decide which control styles best represent the instrument commands and options. Typically, you can categorize instrument commands into three types of control styles: Boolean, digital numeric, and text or ring numeric.

For example, you can represent any instrument command that has two options (such as TRIG:MODE:AUTO | NORMAL) on the front panel with a Boolean switch. In this case, 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), use a text ring rather than a digital numeric because the text ring labels each numeric value with the command it represents. Any command requiring a numeric parameter whose value varies over a wide range and cannot be better represented by a ring can be represented with a digital numeric.

The three control types of Boolean, numeric, and text ring represent most instrument commands on the front panel of your VIs. In addition, block diagram string icons specifically designed for use with these controls exist. These features can simplify string formatting and appending instrument commands into command messages, as discussed in the Assembling Command Strings and Block Diagram topics.

Required Front Panel Controls

In addition to the controls required to operate the instrument, your front panel must also have the following controls:



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 all configuration information necessary to perform the I/O.

dup VISA session output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error in describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See Error Reporting for more information on **error in** clusters.

error out is a cluster containing error information. If **error in** indicates an error, the **status**, **code**, and **source** elements of **error out** have the same values as the corresponding elements of **error in**. If **error in** does not indicate an error, **error out** describes the error encountered by the VI. Refer to the Instrument Driver Error Codes topic for a description of the possible error codes. See Error Reporting for more information on **error out** clusters.

To gain consistency with other LabVIEW instrument drivers, place the **VISA session** control and **dup VISA session** indicator in the upper left and upper right corners of the front panel, and the **error in** and **error out** clusters in the lower left and lower right corners, respectively.

Note: **You can place the error in cluster outside the panel window because it has no interactive meaning and is only needed for programmatic use.**

Control Guidelines

When placing controls on your front panels, use the following style guidelines to ensure uniformity with other LabVIEW VI front panels:

- Use the default font (application) for all LabVIEW instrument driver front panel control labels.The application font is available on all platforms that are using LabVIEW.
- Use **bold** text for control name labels that denote important or primary controls, and reserve plain text for secondary controls.

Note: **In most cases, all instrument driver controls are primary and require bold text. If you are finding yourself placing many secondary or auxiliary controls on panels, this may indicate the need to subdivide your VI into two or more VIs.**

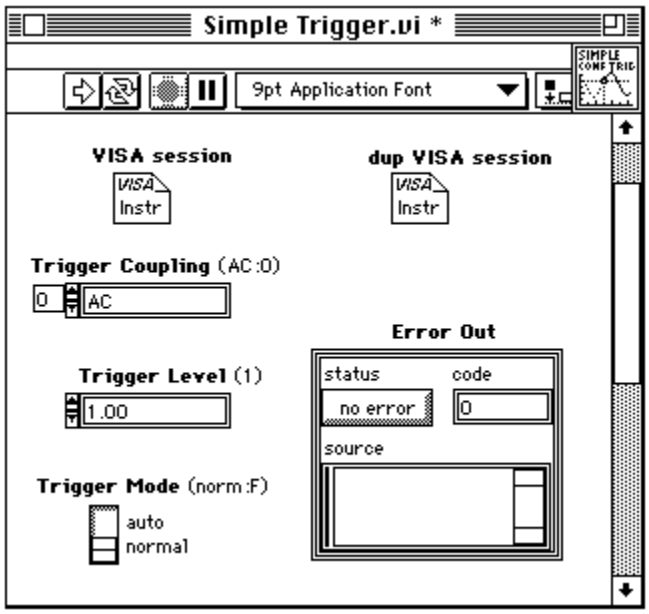
- Use lowercase letters in all words, except abbreviations or acronyms, which require caps (such as ID or GPIB)
- Enclose control default information in parentheses in the control name.

  By including default information in the control name, you can later access that information through the help window. This feature is helpful when you are using the VI in higher-level applications.

  For example, label a function selector ring control whose default is DC volts as item zero **Function** (DCV:0), and label a Boolean mode switch that defaults to true indicating automatic **Mode** (auto:T).

(Notice that the default information is in plain text)**.**

The following figure shows the simple trigger VI after modification to meet the style guidelines:



Block Diagram

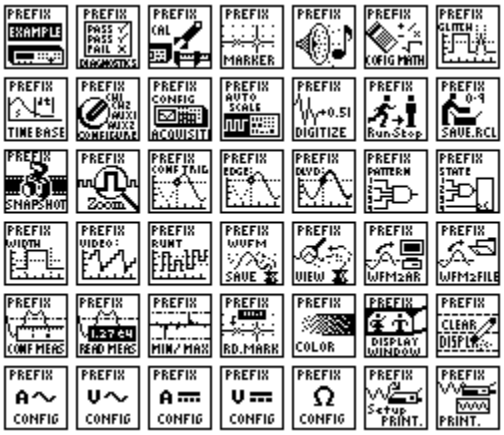
Proper wiring style improves the diagram appearance and eases understanding. The following are recommendations for developing your instrument driver block diagrams:

- Add text labels to each frame of Case and Sequence structures.
- Label control and indicator nodes with normal text.
- Use bold text to make your free label comments stand out.
- Leave room for labels and wires. Do not crowd the diagram. Do not cover wires with loops, cases, labels, or other diagram objects.
- Reduce the number of bends in the wires by aligning data terminals whenever possible. You can use the cursor keys to move objects one pixel width at a time. Use the **Align** and **Distribute** options in the **Edit** menu to add symmetry and straight lines to your diagram.
- Label long wires and complex operations to increase understandability.

Icon

When you use an instrument driver VI programmatically, the icon graphically represents the function (much like the function name of a C library call). Use meaningful icons for every VI. Include text in the icon that identifies 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 only.

You can borrow icons from similar VIs in other instrument drivers. The following figure shows some examples of icons. These sample icons are available in the file `insticon.llb`.



Connector Pane

When you use an instrument driver programmatically, the connector pane defines how to pass parameters to and from the VI. Use the following rules when creating your instrument driver connector panes:

- Place the **VISA session** input and **dup VISA session** output in the upper left and upper right terminals of the LabVIEW instrument driver connector pane.
- Place the **error in** and **error out** clusters in the lower left and lower right terminals of the LabVIEW instrument driver connector pane respectively.
- Place inputs on the left and outputs on the right of the connector pane whenever possible. This promotes a left-to-right data flow when the VI is used in a block diagram.

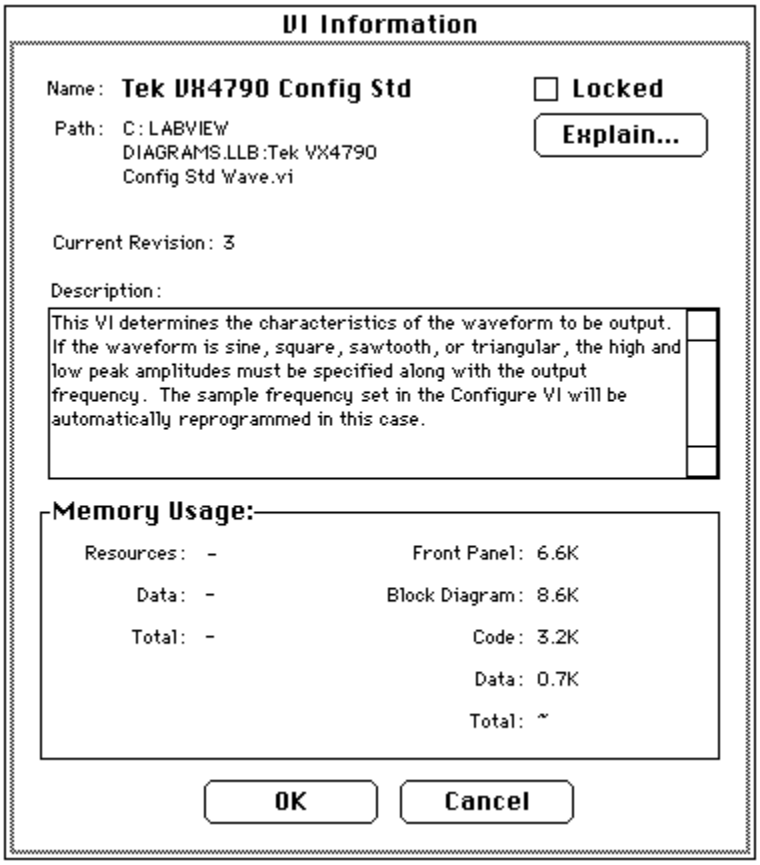
Note: **It is acceptable to choose a connector pane pattern that has extra terminals in case you make unforeseen control or indicator additions to your instrument driver VIs in the future. This procedure prevents you from having to change the pattern and replace all instances of calls to a modified VI.**

Online Help Information

LabVIEW has two types of help mechanisms available to users: a VI Description Dialog Box and Control and indicator Descriptions. You should implement both VI Descriptions and Control Descriptions for all LabVIEW instrument driver VIs and controls that you develop.

VI Description Dialog Box

Users can access VI Description help from the description box of the information window by selecting **Windows**»**Show VI Info...**, as shown in the following figure.

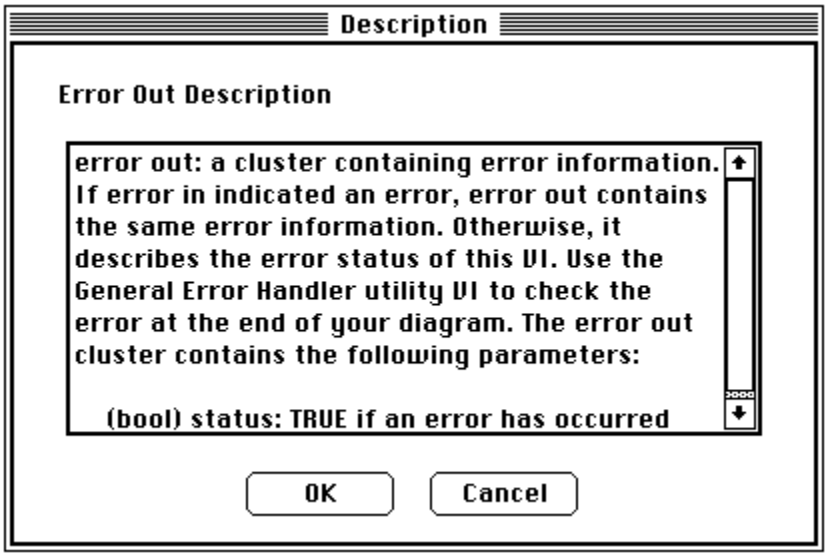


This dialog box should contain the following information:

- A general description of the instrument driver VI
- Control usage rules
- VI interaction with other instrument driver VIs
- Important information concerning the use of the VI

Control and Indicator Descriptions

Control and indicator help is the information most frequently viewed by the user. You can obtain control or indicator help by selecting **Data Operations**»**Description...** from the control or indicator pop-up menu, as shown in the following figure:



The control and indicator help information should contain the following:

- Name of the parameter
- Brief description of the parameter
- Valid range
- Default value
- Interaction with other controls

Be sure to include information showing index numbers and corresponding settings for all ring and slide controls, and settings corresponding to True/False positions on Boolean controlsinstrument driver development:online help information.

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. Consider the following points when developing application VIs for your instrument driver:

- Concentrate on building simple, quality examples that can serve as general models for users. It is not necessary to make your application VIs perform every function found in your instrument driver.
- Build the instrument driver top-level examples from the instrument driver component VIs, and perform common test- and measurement-oriented operations for this particular instrument.
- Do not use the instrument driver application VIs to call the initialize or close instrument driver VIs, because doing so will make the application VIs less useful to higher level applications.

LabVIEW Instrument Driver Standards Checklist

All LabVIEW instrument drivers should conform to recommendations for programming style, error handling, front panels, block diagrams, and online help as described in this topic. Use the following checklist to verify that your instrument driver complies with library standards:

General Issues
Files Created
LabVIEW Instrument Driver Archive- .LLB

VI Front Panels
Icon/Connector Pane/VI Setup
Block Diagram

General Issues

____ All VIs designed for programmatic use; no pop-up VIs or dialog boxes; no interactive input; all controls wired to connector pane. (Soft front panel VIs are allowed only as additional examples built upon the driver VIs; the driver VIs themselves must be for programmatic use.)

____ All VIs are multi-instance: no uninitialized shift registers; no global storage VIs unless specifically designed to work with multiple instruments simultaneously.

____ All VIs fully documented including **Show VI Info...** and control descriptions.

____ Driver uses VISA for all instrument I/O.

____ Driver follows instrument internal driver model: Initialize, Configure, Action/Status, Data, Utility, and Close VIs; also Getting Started, Application, and VI Tree VIs.

____ All VIs use error I/O clusters.

Files Created

____ All VIs in one or more LabVIEW instrument driver archives (`*.llb`).

____ All `.llb` files must have instrument prefix in file name (abbreviate as necessary, for example: `FL45conf.llb`, `FL45trig.llb`, or `FL45meas.llb`).

____ Text file allowed with name in the form `prefix.txt` (example: `FL45.txt`).

____ All `.llb` and `.txt` files placed in directory named with instrument prefix.

____ Instrument directory and files compressed into `prefix.zip` using PKzip (for example, `FL45.zip`).

LabVIEW Instrument Driver Archive - .LLB

____ All VIs saved with meaningful names including instrument prefix and description; no special characters; use Initial Cap form (for example, Fluke 45 Read Measurement).

____ **Functions** and **Controls** palettes well organized, and names entered.

____ Getting Started, Application, and VI Tree VIs given top-level status.

____ Support VIs hidden from the **Functions** palette.

VI Front Panels

____ Contain **VISA session**, **dup VISA session**, **error in**, and **error out** controls.

____ Front panel **Show VI Info...** description is complete.

____ VI History is updated with comments as needed.

____ Controls and Indicators:

- ____ All descriptions are complete. Include defaults, range, items in rings, etc.
- ____ Place labels at upper left of controls; color labels transparent; use the **Size to Text** feature.
- ____ Capitalize initial letters of control names; use the Application font; bold for primary controls (most) and plain for secondary controls (rarely needed).
- ____ Proper defaults are set for each control and default included in name (example: ring control with default DC Volts selection at value zero: Function Select (DCV:0)).
- ____ The proper display format is used, such as hexadecimal for status registers, etc.

____ Use color sparingly or standard gray.

Icon/Connector Pane/VI Setup

____ Create meaningful icons for all VIs.

- ____ Place instrument prefix in upper 1/3 and description text in lower 1/3 of icon.
- ____ Try to keep a common theme for all VIs of a particular driver (if nothing else, use an image of the instrument).
- ____ Create icons for Black & White (required), 16-Color and 256-Color (optional).

____ Select appropriate connector pane.

- ____ Try to keep inputs on left & top; outputs on right & bottom.
- ____ Instrument handle in/out on upper left/right terminal; Error in/out on lower left/right terminal.
- ____ Extra terminals are acceptable for future modifications.

____ Use VI Setups and window options carefully; do not create pop-up panels.

Block Diagram

- ____ Use bold text labels (Application font) to describe each frame of Case and Sequence structures; left-justify; color borders transparent.
- ____ Use plain text labels (Application font) for controls/indicators; place beneath control; left-justify; color borders transparent.
- ____ Dont crowd diagram or cover wires with labels, objects, structures, etc.
- ____ Try for general left-right, top-down data flow.
- ____ Use proper Error I/O wiring techniques; use correct error codes for error reporting.
- ____ Save diagrams with first (or most important) frames or cases showing.

Instrument Driver Template VI Descriptions

Click here to access the Instrument Driver Template VI Overview topic.

This topic describes the Instrument Driver Template VIs.

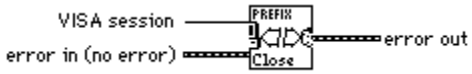
Instrument Driver Template VIs

PREFIX Close
PREFIX Error Message
PREFIX Error Query
PREFIX Initialize
PREFIX Message-Based Template
PREFIX Register-Based Template
PREFIX Reset
PREFIX Revision Query
PREFIX Self-Test

These VIs are located in `examples\instr\insttmpl.llb`.

PREFIX Close

The PREFIX Close VI is a template for creating a Close VI for your particular instrument. It terminates communication with the instrument and deallocates system resources.



Click here to access more information on using Close VIs.

Modify this VI according to the following instructions:

1. Save a copy of this VI.
a. Select **File**»**Save As...**.
b. Replace the word PREFIX in the name of the VI with the prefix for your instrument. For example, you could name the close VI for the Tektronix VX4790 Arbitrary Waveform Generator the TKVX4790 Close VI.
c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you have not already made this `.llb`, you can create it while saving the VI.
2. Edit the icon pane on the front panel.
a. Open the icon editor by double-clicking on the icon pane.
b. Change the word PREFIX to the prefix for your instrument.
c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.
3. Edit the block diagram. Replace the word PREFIX in the string constant labeled VI name with the prefix for your instrument.
4. Edit **Show VI Info...** and all control and indicator description information.
5. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

VISA session input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O. The Prefix Close VI terminates communication with a device and therefore does not have **a dup VISA session**

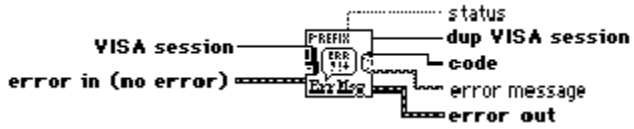
output.

error in describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces.. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

PREFIX Error Message

The PREFIX Error Message VI is a template for creating an Error Message VI for your particular instrument. It translates the error status information returned from a LabVIEW instrument driver VI to a user-readable string.



Modify this VI according to the following instructions:

1. Save a copy of this VI.
   a. Select **File**»**Save As...**.
   b. Replace the word PREFIX in the name of the VI with the prefix for your instrument. For example, you could name the error message VI for the Tektronix VX4790 Arbitrary Waveform Generator the TKVX4790 Error Message VI.
   c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you have not already made this `.llb`, you can create it while saving the VI.
2. Edit the icon pane on the front panel.
   a. Open the icon editor by double-clicking on the icon pane.
   b. Change the word PREFIX to the prefix for your instrument.
   c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.
3. Edit the block diagram. Replace the word PREFIX in the string constant labeled VI name with the prefix for your instrument.
4. Edit the **user-defined codes** and **user-defined descriptions** array controls that are located on the front panel. Enter instrument specific error codes in the user-defined codes array and enter their corresponding descriptions in the user-defined descriptions array.
5. Select **Operate**»**Make Current Values Default**.
6. Edit **Show VI Info...** and all control and indicator description information.
7. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

VISA session input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error in describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters.

dup VISA session output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

code contains the error number from the error in input cluster.

error out contains error information. If **error in** indicates an error, then **error out** contains the

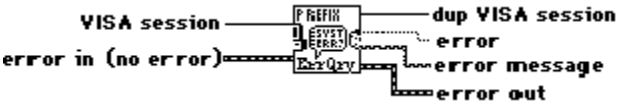
same error information. Otherwise it describes the error status that this VI produces. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

status contains the status from the **error in** input cluster.

error message returns a user-readable string based on the **error in** input cluster.

PREFIX Error Query

The PREFIX Error Query VI is a template for creating an error query VI for your particular instrument. It queries the instrument and returns instrument-specific error information.



Modify this VI according to the following instructions:

1. Save a copy of this VI.
a. Select **File**»**Save As...**.
b. Replace the word PREFIX in the name of the VI with the prefix for your instrument. For example, you could name the error query VI for the Tektronix VX4790 Arbitrary Waveform Generator the TKVX4790 Error Query VI.
c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you have not already made this `.llb`, you can create it while saving the VI.
2. Edit the icon pane on the front panel.
a. Open the icon editor by double-clicking on the icon pane.
b. Change the word PREFIX to the prefix for your instrument.
c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.
3. Edit the block diagram.
a. Replace the word PREFIX in the string constant labeled VI name with the prefix for your instrument.
b. Edit the string constant labeled error query command by entering the reset command for your instrument.
4. Edit **Show VI Info...** and all control and indicator description information.
5. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

VISA session input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error in describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters.

dup VISA session output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error is the error which has occurred.

error message contains any error messages from the instrument.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

PREFIX Initialize

The PREFIX Initialize VI is a template for creating an Initialize VI for message-based instruments. The function is used to establish communication with a remote instrument. Also the VI can perform user selectable ID query and/or reset operations.

Click here to access more information on using Initialize VIs.

Modify this VI according to the following instructions:

1. Save a copy of this VI.

a. Select **File**»**Save As...**.

b. Replace the word PREFIX in the name of the VI with the prefix for your instrument. For example, you could name the Initialize VI for the Tektronix VX4790 Arbitrary Waveform Generator the TKVX4790 Initialize VI.

c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you have not already made this `.llb`, you can create it while saving the VI.

2. Edit the icon pane on the front panel.

a. Open the icon editor by double-clicking on the icon pane.

b. Change the word PREFIX to the prefix for your instrument.

c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.

3. Edit the block diagram.

a. Replace the word PREFIX in the string constant labeled VI name with the prefix for your instrument.

b. Set the value of ID Query command to be the ID Query for your instrument. Change the constant labeled ID Query Response to be the ID Query response for your instrument. If your instrument driver is meant to work with multiple instruments, place all possible ID Query responses in the string constant separated by carriage returns.

4. Replace the PREFIX Reset VI with the name of the reset VI for your instrument. If you have not created a reset VI for your instrument, load the PREFIX Reset VI in the file `instrtmp.llb` by selecting **File**»**Open...** and following the instructions on the front panel.

5. Change the string constant labeled default setup string by entering the command string you want to send to the instrument.

6. Edit **Show VI Info...** and all control and indicator description information.

7. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

TF **instrument descriptor**. Based on the syntax of the **instrument descriptor** input, the INSTR Open VI configures the I/O interface and generates an instrument handle that is used by all other instrument driver functions. The grammar for the **instrument descriptor** is shown below. Optional parameters are shown in square brackets ([ ]).

Interface	**Syntax**
GPIB	GPIB[*board*]::*primary address*[::*secondary address*][::INSTR]
VXI	VXI::*VXI logical address*[::INSTR]
GPIB-VXI	GPIB-VXI[*board*][::*GPIB-VXI primary address*]::*VXI logical*

address[::INSTR]

As shown in the preceding table, the GPIB keyword is used for GPIB instruments. The VXI keyword is used for VXI instruments through either embedded or MXIbus controllers. The GPIB-VXI keyword is used for a National Instruments GPIB-VXI controller.

The following table shows the default values for optional parameters:

Optional Parameter	Default Value
board	0
secondary address	none
GPIB-VXI primary address	1

ID query determines if the Initialize VI will attempt to perform identification query.

TRUE: Perform identification query.

FALSE: Do not perform identification query.

reset determines if the Initialize VI will reset the device.

TRUE: Reset the device.

FALSE: Do not reset the device.

error in describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters.

VISA session output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

PREFIX Message-Based Template

The PREFIX Message-Based Template VI is a template for creating a message-based VI for your particular instrument.

Modify this VI according to the following instructions:

1. Save a copy of this VI.
   a. Select **File**»**Save As...**.
   b. Rename the VI substituting the prefix for your instrument and a unique name describing the VIs purpose.
   c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you have not already made this `.llb`, you can create it while saving the VI.

2. Edit the front panel. Add controls and indicators as necessary to perform the VI function.
3. Edit the icon pane on the front panel.

a. Open the icon editor by double-clicking on the icon pane.

b. Change the word PREFIX to the prefix for your instrument.

c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.

4. Edit the icon connector pane on the front panel.

a. Pop-up on the icon pane and select **Show Connector**.

b. Pop-up on the connector pane and select **Patterns**. Choose a pattern suitable for the controls and indicators on the front panel. Remember the instrument handle control goes in the upper left-hand corner, and the error in and error out clusters go in the lower left-hand and right-hand corners, respectively.

5. Edit the block diagram. Replace the word PREFIX in the string constant labeled `VI name` with the prefix for your instrument.
6. Use the String functions found in the **Functions** palette to build a command string based on the input controls. Wire the resulting command string to the write buffer control of the INSTR Send Message VI.
7. If the instrument generates a response, use the INSTR Receive Message VI to read from the instrument. Use the String functions found in the Functions palette to parse the response and wire it to the indicator terminals.
8. Edit **Show VI Info...** and all control and indicator description information.
9. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

VISA session input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

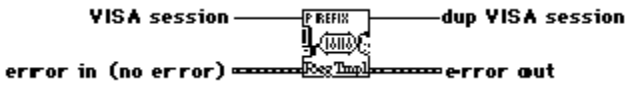
error in describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters..

dup VISA session output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

PREFIX Register-Based Template

The PREFIX Register-Based Template VI is a template for creating a register-based VI for your particular instrument.



Modify this VI according to the following instructions:

1. Save a copy of this VI.

a. Select **File**»**Save As...**.

b. Rename the VI substituting the prefix for your instrument and a unique name describing the VIs purpose.

c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you

have not already made this `.llb`, you can create it while saving the VI.

2. Edit the front panel. Add controls and indicators as necessary to perform the VI function.
3. Edit the icon pane on the front panel.

a. Open the icon editor by double-clicking on the icon pane.

b. Change the word PREFIX to the prefix for your instrument.

c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.

4. Edit the icon connector pane on the front panel.

a. Pop-up on the icon pane and select **Show Connector**.

b. Pop-up on the connector pane and select **Patterns**. Choose a pattern suitable for the controls and indicators on the front panel. Remember the instrument handle control goes in the upper left-hand corner, and the error in and error out clusters go in the lower left-hand and right-hand corners, respectively.

5. Edit the block diagram.

a. Replace the word PREFIX in the string constant labeled VI name with the prefix for your instrument.

b. Change the enumerated type labeled `memory space` to be the memory space you plan to access, and change the constant labeled `base address` to be the base address of your instrument.

c. Change the constants labeled `register offset` to be the registers that you plan to peek or poke. Add or remove the VISA Peek 16 function or the VISA Poke 16 function, as necessary. If you want to access A24 or A32 addresses, replace these functions with the appropriate VIs from the LabVIEW Instrument Driver Support Library.

6. Edit **Show VI Info...** and all control and indicator description information.
7. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

 **VISA session** input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

 **error in** describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters.

 **dup VISA session** output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

 **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

PREFIX Reset

The PREFIX Reset VI is a template for creating a reset VI for your particular instrument. The Reset VI places the instrument in a default state.

Modify this VI according to the following instructions:

1. Save a copy of this VI.

a. Select **File**»**Save As...**.

b. Replace the word PREFIX in the name of the VI with the prefix for your instrument. For example, you could name the reset VI for the Tektronix VX4790 Arbitrary Waveform Generator the TKVX4790 Reset VI.

c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you have not already made this `.llb`, you can create it while saving the VI.

2. Edit the icon pane on the front panel.

a. Open the icon editor by double-clicking on the icon pane.

b. Change the word PREFIX to the prefix for your instrument.

c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.

3. Edit the block diagram.

a. Replace the word PREFIX in the string constant labeled `VI name` with the prefix for your instrument.

b. Edit the string constant labeled `Reset Command` by entering the reset command for your instrument.

4. Edit **Show VI Info...** and all control and indicator description information.

5. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

VISA session input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

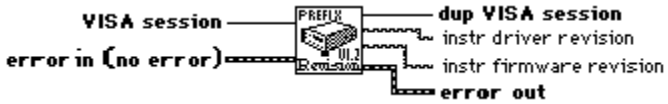
error in describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters.

dup VISA session output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

PREFIX Revision Query

The PREFIX Revision Query VI is a template for creating a Revision Query VI for your particular instrument. It returns the revision of the instrument driver and the firmware revision of the instrument.



Modify this VI according to the following instructions:

1. Save a copy of this VI.

a. Select **File**»**Save As...**.

b. Replace the word PREFIX in the name of the VI with the prefix for your instrument. For example, you could name the revision VI for the Tektronix VX4790 Arbitrary Waveform Generator the TKVX4790 Revision VI.

c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you have not already made this `.llb`, you can create it while saving the VI.

2. Edit the icon pane on the front panel.

a. Open the icon editor by double-clicking on the icon pane.

b. Change the word PREFIX to the prefix for your instrument.

c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.

3. Edit the block diagram.

a. Replace the word PREFIX in the string constant labeled `VI name` with the prefix for your instrument.

b. Edit the string constant labeled `revision query` command by entering the revision command for your instrument.

4. Edit **Show VI Info...** and all control and indicator description information.

5. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

VISA session input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error in describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters.

dup VISA session output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

error out contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

instr driver revision contains the revision level of the instrument driver source code.

instr firmware revision contains the firmware revision level of the instrument.

PREFIX Self-Test

The PREFIX Self-Test VI is a template for creating a self-test VI for your particular instrument. It tells the instrument to perform a self-test and returns the result.

Modify this VI according to the following instructions:

1. Save a copy of this VI.

a. Select **File**»**Save As...**.

b. Replace the word PREFIX in the name of the VI with the prefix for your instrument. For example, you could name the self-test VI for the Tektronix VX4790 Arbitrary Waveform Generator the TKVX4790 Self-Test VI.

c. Save the VI in the file `prefix.llb` where `prefix` is the prefix for your instrument. If you have not already made this `.llb`, you can create it while saving the VI.

2. Edit the icon pane on the front panel.

a. Open the icon editor by double-clicking on the icon pane.

b. Change the word PREFIX to the prefix for your instrument.

c. Edit the icon for all three color modes: Black and White, 16-color, and 256-color.

3. Edit the block diagram.

a. Replace the word PREFIX in the string constant labeled `VI name` with the prefix for your instrument.

b. Edit the string constant labeled `self-test command` by entering the self-test command for your instrument.

4. Edit **Show VI Info...** and all control and indicator description information.
5. Delete the Modification Instructions text box from the front panel, resize the front panel, and save this VI.

 **VISA session** input is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

 **error in** describes error conditions that occur before this VI executes. The default input of this cluster is `no error`. See the Error Reporting topic for a description of the parameters in the **error in** clusters.

 **dup VISA session** output is a unique identifier reference to a device I/O session. It identifies the device with which the VI communicates and all configuration information necessary to perform the I/O.

 **self-test error** returns the error code from the instrument.

 **self-test response** contains the result of the instrument driver self-test.

 **error out** contains error information. If **error in** indicates an error, then **error out** contains the same error information. Otherwise it describes the error status that this VI produces. See the Error Reporting topic for a description of the parameters in the **error out** clusters.

Instrument Driver Template VI Overview

Click here to access the Instrument Driver Template VI Descriptions topic.

The LabVIEW instrument driver templates are the foundation for all LabVIEW instrument driver development.

Instrument Driver Template VIs

Instrument Driver Template VIs

The template VIs have a simple, flexible structure and a common set of instrument driver VIs that you can use for driver development. The templates establish a standard format for all LabVIEW drivers and each has instructions for modifying it for a particular instrument.

The LabVIEW instrument driver templates are predefined instrument driver VIs that perform common operations such as initialization, self- test, reset, error query, and so on. Instead of developing your own VIs to accomplish these tasks, you should use the LabVIEW instrument driver template VIs, which already conform to the LabVIEW standards for instrument drivers.

The following table lists the Instrument Driver Template VIs:

Instrument Driver Template VI	Description
PREFIX Init	Establishes communication with the instrument and performs user selectable identification query and reset operations.
PREFIX Close	Terminates communication with the instrument.
PREFIX Revision	Queries the instrument driver and instrument firmware revisions.
PREFIX Self-Test	Performs an instrument self-test and returns the result.
PREFIX Reset	Places the instrument in a known state.
PREFIX Error Query	Queries the instrument for errors.
PREFIX Error Message	Converts error numbers from other instrument driver functions to user-readable strings.
PREFIX Message-Based Template	A template for creating instrument driver VIs for a message-based instrument.
PREFIX Register-Based Template	A template for creating instrument driver VIs for a register-based instrument.

Rather than developing your own VIs to accomplish these tasks, you should use the LabVIEW instrument driver template VIs which already conform to the LabVIEW standards for instrument drivers. The template VIs are IEEE 488.2 compatible and work with IEEE 488.2 instruments with minimal modifications. For non-IEEE 488.2 instruments, use the template VIs as a shell or pattern, which you can modify by

substituting your corresponding instrument-specific commands where applicable. After modifying the VIs, you will have the base level driver that implements all of the template instrument driver VIs for your particular instrument.

Additionally, LabVIEW instrument drivers developed from the template VIs will be similar to other instrument drivers in the library. Therefore, you will have a higher level of familiarity and understanding when you work with multiple instrument drivers.

Initialize
Close
Reset
Self-Test
Error Query and Error Message
Revision Query
Message-Based Template and Register-Based Template
Example

Initialize

The Initialize VI is the first VI called when you are accessing an instrument driver. It configures the communications interface, manages handles, and sends a default command to the instrument. Typically, the default setup configures the instrument operation for the rest of the driver (including turning headers on or off, or using long or short form for queries). After successful operation, the Initialize VI returns a VISA session that addresses the instrument in all subsequent instrument driver VIs.

The VI has an instrument descriptor string as an input. Based on the syntax of this input, the VI configures the I/O interface and generates an instrument handle for all other instrument driver VIs. The following table shows the grammar for the instrument descriptor. Optional parameters are shown in square brackets ([]).

Interface	**Syntax**
GPIB	GPIB[*board*]::*primary address*[::*secondary address*] [::INSTR]
VXI	VXI::*VXI logical address*[::INSTR]
GPIB-VXI	GPIB-VXI[*board*][::*GPIB-VXI primary address*]::*VXI logical address*[::INSTR]

The GPIB keyword is used with GPIB instruments. The VXI keyword is used for either embedded or MXIbus controllers. The GPIB-VXI keyword is used for a National Instruments GPIB-VXI controller.

Additionally, the Initialize VI can perform selectable ID query and reset operations. In other words, you can disable the ID query when you are attempting to use the driver with a similar but different instrument without modifying the driver source code. Also, you can enable or disable the reset operation. This feature is useful for debugging when resetting would take the instrument out of the state you were trying to test.

Close

All LabVIEW instrument drivers should include a Close VI. The Close VI is the last VI called when controlling an instrument. It terminates the software connection to the instrument and deallocates system resources. Additionally, you can choose to place the instrument in an idle state. For example, if you are developing a switch driver, you can disconnect all switches when closing the instrument driver.

Reset

All LabVIEW instrument drivers have a Reset VI that places the instrument in a default state. The default state that the Reset VI places the instrument in should be documented in the help information for the Reset VI. In an IEEE 488.2 instrument, this VI sends the command string *RST to the instrument. When you reset the instrument from the Initialize VI, this VI is called. Also, you can call the Reset VI separately.

Self-Test

If an instrument has self-test capability, the LabVIEW instrument driver should contain a Self-test VI to instruct the instrument to perform a self-test and return the result of that self-test.

Error Query and Error Message

If an instrument has error query capability, the LabVIEW instrument driver has Error Query and Error Message VIs. The Error Query VI queries the instrument and returns the instrument-specific error information. The Error Message VI translates the error status information returned from a LabVIEW instrument driver VI into a user-readable string.

Revision Query

LabVIEW instrument drivers have a Revision Query VI. This VI outputs the following:

- The revision of the instrument driver.
- The firmware revision of the instrument being used (If the instrument firmware revision cannot be queried, the Revision Query VI should return the literal string `Not Available`.)

Message-Based Template and Register-Based Template

The Message-Based and Register-Based template VIs are the starting point for developing your own instrument driver VIs. The template VIs have all required instrument driver controls, and instructions for modification for a particular instrument.

Example

The following table shows the LabVIEW VI organization of the instrument driver for the Tektronix VX4790 Arbitrary Function Generator. The hierarchy shows the developer specified VIs derived from the instrument manual as well as the template VIs.

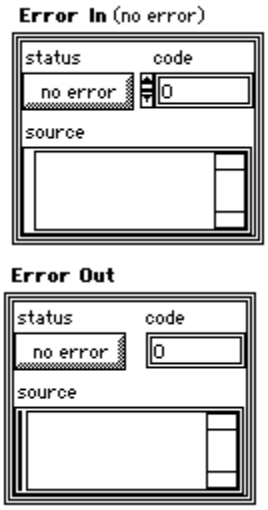
Level	Name	Type
Application VIs	Output Standard Waveform	(Developer Specified)
	Output Arbitrary Waveform	(Developer Specified)
Component VIs	Initialize VI	(Template)
	Configuration VIs	
	Configure Setup	(Developer
	Configure Sample Frequency	Specified)
	Configure Voltage / Attenuation	(Developer

	Configure Standard Waveform	Specified) (Developer Specified) (Developer Specified)
	Actions VIs	
	Run / Stop	(Developer Specified)
	Data Functions	
	Download Arbitrary Waveform	(Developer Specified)
Component VIs	Utilities VIs	
	Reset	(Template)
	Self-Test	(Template)
	Revision Query	(Template)
	Error Query	(Template)
	Error Message	(Template)
	Send Message	(Template)
	Receive Message	
	Close VI	(Template)

Error Reporting

VISA Error In/Error Out
GPIB Error In/Error Out

LabVIEW instrument drivers use error clusters to report all errors, as shown in the following figure.



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 all the way through your instrument driver and out to your full application.

Another benefit of error input/output is that data dependency is added to VIs that are not otherwise data dependent.

error in describes error conditions that occur before this function or 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 function or VI does not perform an operation. Instead it places the value of the **error in** cluster in 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 Instrument Driver Error Codes topic for a description of the possible error codes.

source is the name of the function or VI that produced the error.

error out is a cluster containing error information. If **error in** indicates an error, the **status**, **code**, and **source** elements of **error out** have the same values as the corresponding elements of **error in**. If **error in** does not indicate an error, **error out** describes the error encountered by the function or VI.

***Note*: If you use an instrument driver VI in a While Loop, you should stop the loop if the status Boolean in the error out cluster is TRUE. You can also wire the error cluster to the General Error Handler VI, which can then decipher the error information and inform the user.**

The General Error Handler VI is in Functions»Time & Dialog. For more information on this VI, see the General Error Handler topic. Refer to the Instrument Driver Error Codes topic for a description of possible error codes.

VISA Error In/Error Out

error in and **error out** terminals comprise the error clusters in each VISA function. The error cluster contains three fields. The **status field** is a Boolean which is TRUE when an error occurs, FALSE when no error occurs. The **code field** will be a VISA error code value if an error occurs during a VISA function. See the VISA Error Codes topic for more information. The **source field** is a string which describes where the error has occurred. By wiring the **error out** of each function to the **error in** of the next function, the first error condition is recorded and propagated to the end of the diagram where it is reported in only one place.

GPIB Error In/Error Out

error in and **error out** terminals comprise the error clusters in each traditional GPIB function. The error cluster contains three fields. The **status** field is a Boolean which is TRUE when an error occurs, FALSE when no error occurs. The **code field** will be a GPIB error code value if an error occurs during a GPIB function. The **source field** is a string which describes where the error has occurred. If the **status field** of the **error in** parameter to a function is set, the function is not executed and the same error cluster is passed out. By wiring the **error out** of each function to the **error in** of the next function, the first error condition is recorded and propagated to the end of the diagram where it is reported in only one place.

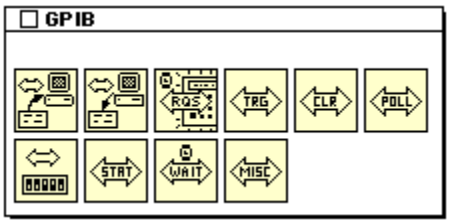
Traditional GPIB Function Descriptions

This topic describes the traditional GPIB functions. You use these functions to handle situations outside the scope of the IEEE 488.2 standard.

Click here to access the GPIB Overview topic

The following figure shows the **GPIB** palette which you access by selecting **Functions»Instrument I/O»GPIB**:

Click on one of the icons below for VI description information. You can also click on one of the text jumps below the icons to access VI descriptions.

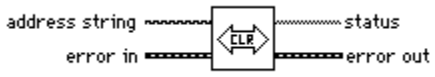


GPIB Clear
GPIB Initialization
GPIB Misc
GPIB Read
GPIB Serial Poll
GPIB Status
GPIB Trigger
GPIB Wait
GPIB Write
Wait for GPIB RQS

For examples of how to use the traditional GPIB functions, see `examples\instr\smplgpib.llb`.

GPIB Clear

Sends either SDC (Selected Device Clear) or DCL (Device Clear).



address string. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

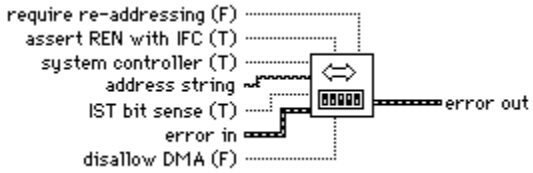
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

GPIB Initialization

Configures the GPIB interface at address string.



require re-addressing. If **require re-addressing** is TRUE, the function addresses the device before every read or write. If FALSE, the device must be able to retain addressing from one read or write to the next.

assert REN with IFC. If **assert REN with IFC** is TRUE, and if this Controller (specified by the ID in address string) is the System Controller, the function asserts the Remote Enable line.

system controller. If **system controller** is TRUE, this Controller acts as the System Controller.

address string. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

IST bit sense. If **IST bit sense** is TRUE, the Individual Status bit of the device responds TRUE to a parallel poll; if **IST bit sense** is FALSE, the Individual Status bit of the device responds FALSE to a parallel poll.

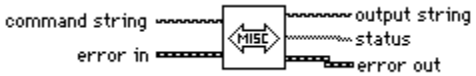
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

disallow DMA. If **disallow DMA** is TRUE, this device uses programmed I/O for data transfers.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

GPIB Misc

Performs the GPIB operation indicated by command string. Use this low-level function when the previously described high-level functions are not suitable.



command string. The following table shows the functions you can specify in command string. See the GPIB Device and Controller Functions topic for more information.

Command String Functions Table

Device Functions	Description
loc *address*	Go to local.
off *address*	Take device offline.
pct *address*	Pass control.
ppc *byte address*	Parallel poll configure (enable or disable).
cac 0/1	Become active Controller.
cmd *string*	Send IEEE 488 commands.
dma 0/1	Set DMA mode or programmed I/O mode.

gts 0/1	Go from active Controller to standby.
ist 0/1	Set individual status bit.
llo	Local lockout.
loc	Place Controller in local state.
off	Take controller offline.
ppc *byte*	Parallel poll configure (enable or disable).
ppu	Parallel poll unconfigure all devices.
rpp	Conduct parallel poll.
rsc 0/1	Request or release system control.
rsv *byte*	Request service and/or set the serial poll status byte.
sic	Send interface clear.
sre 0/1	Set or clear remote enable.

To specify the GPIB Controller used by this function, use a command string in the form ID: xxx, where ID is the GPIB Controller (bus number) and xxx is the three-character command and its corresponding arguments, if any. If you do not specify a Controller ID, LabVIEW assumes 0.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

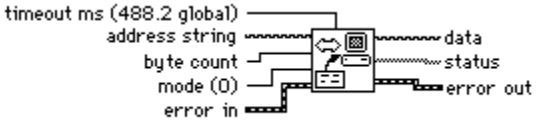
output string is the response the function returns when you conduct a parallel poll with the rpp command string.

status describes the state of the GPIB Controller. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

GPIB Read

Reads byte count number of bytes from the GPIB device at address string.



timeout ms. The operation aborts if it does not complete within **timeout ms**. If a timeout occurs, bit 14 of **status** is set. To disable timeouts, set **timeout ms** to 0. To use the 488.2 global timeout, leave this input unwired.

You use the SetTimeOut function to change the default value (the 488.2 global timeout) of **timeout ms**. Initially, **timeout ms** defaults to 25,000. See the description of the SetTimeOut function for more information.

address string. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

byte count specifies the number of bytes the function reads from the GPIB device.

mode specifies conditions other than reaching byte count for terminating the read.

0: No EOS character. The EOS termination mode is disabled.
1: EOS character is CR. Read terminated on EOI, **byte count**, or CR.
2: EOS character is LF. Read terminated on EOI, **byte count**, or LF.
x: Any other mode indicates the number (decimal) of the desired EOS character. See the Multiline Interface Messages topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

data string. The function returns the data read in **data string**.

status. See the status topic for more information.

The GPIB Read Function terminates when it:

Reads the number of bytes requested.

Detects an error.

Exceeds the time limit.

Detects the END message (EOI asserted).

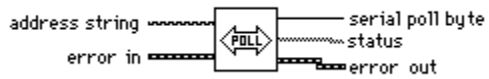
Detects the EOS character (if this option is enabled by the value supplied to **mode**).

Note: **The function compares all eight bits when it checks for the EOS character.**

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

GPIB Serial Poll

Performs a serial poll of the device indicated by address string.



address string is the address of the device the function polls. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in** See GPIB Error In/Error Out for error information specific to GPIB.

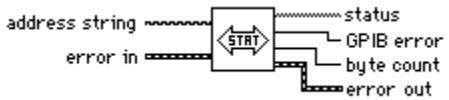
serial poll byte is the response from the device. If the addressed device does not respond within the timeout limit, **serial poll byte** is -1.

status. If the addressed device does not respond within the timeout limit, the function returns GPIB error 6 (bit 14 of **status** is set). See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

GPIB Status

Shows the status of the GPIB Controller indicated by address string after the previous GPIB operation.



address string. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. The following table shows the numeric value and symbolic status of each bit in **status**. This table also includes a description of each bit.

Status Bits Table

Status Bit	Numeric Value	Symbolic Status	Description
0	1	DCAS	Device Clear state
1	2	DTAS	Device Trigger State
2	4	LACS	Listener Active
3	8	TACS	Talker Active
4	16	ATN	Attention Asserted
5	32	CIC	Controller-In-Charge
6	64	REM	Remote State
7	128	LOK	Lockout State
8	256	CMPL	Operation Completed
12	4096	SRQI	SRQ Detected while CIC
13	8192	END	EOI or EOS Detected
14	16384	TIMO	Timeout
15	-32768	ERR	Error Detected

GPIB error contains the most recent error code reported by any of the GPIB functions. The following table shows the possible values for **GPIB error** if bit 15 of **status** is set.

GPIB Error Codes Table

GPIB Error	Symbolic Status	Description
0	EDVR	Error Connecting to Driver
1	ECIC	Command Requires GPIB Controller to be CIC

2	ENOL	Write Detected No Listeners
3	EADR	GPIB Controller Not Addressed Correctly
4	EARG	Invalid Argument or Arguments
5	ESAC	Command Requires GPIB to be System Controller
6	EABO	I/O Operation Aborted
7	ENEB	Non-existent Board
8	EDMA	DMA Hardware Not Detected
9	EBTO	DMA Hardware uP Bus Timeout
11	ECAP	No Capability
12	EFSO	File System Operation Error
13	EOWN	Shareable Board Exclusively Owned
14	EBUS	GPIB Bus Error
15	ESTB	Serial Poll Byte Queue Overflow
16	ESRQ	SRQ Stuck On
17	ECMD	Unrecognized Command
19	EBNP	Board Not Present
20	ETAB	Table Error
30	NADDR	No GPIB Address Input
31	NSTRG	No String Input (Write)
32	NCNT	No Count Input (Read)
61	EPAR	Serial Port Parity Error
62	EORN	Serial Port Overrun
63	EOFL	Port Receive Buffer Overflow
64	EFRM	Serial Port Framing Error
65	SPTMO	Serial Port Timeout, Bytes Not Received at Serial Port

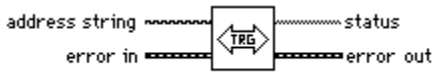
byte count is the number of bytes the previous GPIB operation sent.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error

In/Error Out for error information specific to GPIB.

GPIB Trigger

Sends GET (Group Execute Trigger) to the device indicated by address string.



address string. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

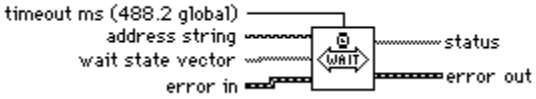
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out** See GPIB Error In/Error Out for error information specific to GPIB.

GPIB Wait

Waits for the state(s) indicated by wait state vector at the device indicated by address string.



timeout ms. The operation aborts if it does not complete within **timeout ms**. If a timeout occurs, the function sets bit 14 of **status**. To disable timeouts, set **timeout ms** to 0. To use the 488.2 global timeout, leave this input unwired.

You use the SetTimeOut function to change the default value (the 488.2 global timeout) of **timeout ms**. Initially, timeout ms defaults to 25,000. See the description of the SetTimeOut function for more information.

address string. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

wait state vector. Bits set to TRUE in **wait state vector** indicate states for which the function waits. If more than one bit is set, the function terminates when any one of the desired states exists.

The following table defines the bits that you can set in wait state vector. This table also lists the numeric value and description of each bit. These bits are the same as the ones that other GPIB functions return; however, only the bits listed are valid for this function.

Wait State Vector Bits Table

Wait State Vector Bit	Numeric Value	Symbolic Status	Description
0	1	DCAS	Device Clear state
1	2	DTAS	Device Trigger State
2	4	LACS	Listener Active
3	8	TACS	Talker Active

4	16	ATN	Attention Asserted
5	32	CIC	Controller-In-Charge
6	64	REM	Remote State
7	128	LOK	Lockout State
12	4096	SRQI	SRQ Detected while CIC
13	8192	END	EOI or EOS Detected
14	16384	TIMO	Timeout

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

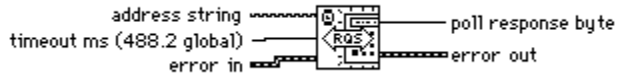
status. If you are waiting for multiple states, check **status** to see which state caused the function to terminate. Refer to the GPIB Status Function for other status bit descriptions. See the status topic for more information.

This function can run in parallel with other functions because LabVIEW alternately checks for **status** and executes other functions. In addition, multiple calls to this function can execute in parallel, so you can wait for different states on different Controllers at the same time or for multiple states to exist.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

Wait for GPIB RQS

Waits for the device indicated by address string to assert SRQ.



timeout ms. The operation aborts if it does not complete within **timeout ms**. If a timeout occurs, poll response byte is -1. To disable timeouts, set **timeout ms** to 0. To use the 488.2 global timeout, leave this input unwired.

You use the SetTimeOut function to change the default value (the 488.2 global timeout) of **timeout ms**. Initially, **timeout ms** defaults to 25,000. See the description of the SetTimeOut function for more information.

address string. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

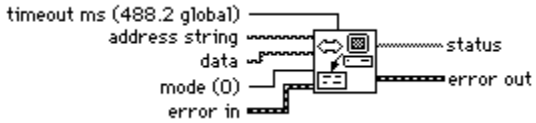
poll response byte. If SRQI is set, the function polls the device at the specified address to see if it requested service. When the specified device requests service (bit 6 is set in poll response byte), the function returns the serial poll response.

If the device indicated by address string does not respond within the timeout limit, **poll response byte** is -1.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

GPIB Write

Writes **data** to the GPIB device identified by **address string**.



timeout ms. The operation aborts if it does not complete within **timeout ms**. If a timeout occurs, bit 14 of **status** is set. To disable timeouts, set **timeout ms** to 0. To use the 488.2 global timeout, leave this input unwired.

You use the SetTimeOut function to change the default value (the 488.2 global timeout) of **timeout ms**. Initially, **timeout ms** defaults to 25,000 See the description of the SetTimeOut function for more information.

address string. If you specify an address in address string, the function sends SDC to that address. If **address string** is unwired, the function sends DCL. See the address string topic for more information.

data is the data the function writes to the GPIB device.

mode indicates how to terminate the GPIB Write.

0: Send EOI with the last character of the string.
1: Append CR to the string and send EOI with CR.
2: Append LF to the string and send EOI with LF.
3: Append CR LF to the string and send EOI with LF.
4: Append CR to the string but do not send EOI.
5: Append LF to the string but do not send EOI.
6: Append CR LF to the string but do not send EOI.
7: Do not send EOI.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

GPIB Device and Controller Functions

This topic describes the functions listed in the GPIB Misc function description. The device functions send configuration information to a specific instrument (device). The controller functions configure the Controller or send IEEE 488 commands to which all instruments respond. Notice that there are both device and Controller versions of the **ppc** and **loc** commands. The syntax and use of the commands are slightly different for each version.

You can use these functions with all GPIB Controllers accessible by LabVIEW, unless stated otherwise in the function description below. An ECMD error (17) results when you execute a function for a GPIB Controller without the specified capability. The function syntax is strict. Each function recognizes only lowercase characters and allows only one space between the function name and the arguments.

Device Functions

locGo to local

offTake device offline
pctPass control
ppcParallel poll configure

loc Go to local

syntax **loc** *address*

loc temporarily moves devices from a remote program mode to a local mode.

address is the GPIB address of the device. This argument indicates both primary and secondary addresses if you use the form *primary+secondary*, where *primary* and *secondary* are the decimal values of the primary and secondary addresses. For example, if *primary* is 2 and *secondary* is 3, then *address* is 2+3.

loc sends the GTL (Go To Local) message to the GPIB device.

off Take device offline

syntax **off** *address*

off takes the device at the specified GPIB address offline. This is only needed when sharing a device with another application which is using the NI 488 GPIB Library.

address is the GPIB address of the device. This argument indicates both primary and secondary addresses if you use the form *primary+secondary*, where *primary* and secondary are the decimal values of the primary and secondary addresses. For example, if *primary* is 2 and *secondary* is 3, then *address* is 2+3.

pct Pass control

syntax **pct** *address*

pct passes Controller-in-Charge (CIC) authority to the device at the specified address. The GPIB Controller automatically goes into an idle state. The function assumes that the device to which pct passes control has Controller capability.

address is the GPIB address of the device. This argument indicates both primary and secondary addresses if you use the form *primary+secondary*, where *primary* and secondary are the decimal values of the primary and secondary addresses. For example, if *primary* is 2 and *secondary* is 3, then *address* is 2+3.

pct sends the following command sequence:

1. Talk address of the device
2. Secondary address of the device, if applicable
3. Take Control (TCT)

ppc Parallel poll configure

syntax **ppc** *byte address*

ppc enables the instrument to respond to parallel polls.

byte is 0 or a valid parallel poll enable (PPE) command. If *byte* is 0, the parallel poll disable (PPD) *byte* 0x70 is sent to disable the device from responding to a parallel poll. Each of the 16 PPE messages selects a GPIB data line (DIO1 through DIO8) and sense (1 or 0) that the device must use when it responds to the Identify (IDY) message during a parallel poll. The device compares the ist sense and

drives the indicated DIO line TRUE or FALSE.

address is the GPIB address of the device. This argument indicates both primary and secondary addresses if you use the form *primary+secondary*, where *primary* and secondary are the decimal values of the primary and secondary addresses. For example, if *primary* is 2 and *secondary* is 3, then *address* is 2+3.

Controller Functions

cacBecome active Controller
cmdSend IEEE 488 commands
dmaSet DMA mode or programmed I/O mode
gtsGo from active Controller to standby
istSet individual status bit
lloLocal lockout
locPlace Controller in local state
offTake controller offline
ppcParallel poll configure (enable and disable)
ppuParallel poll unconfigure
rppConduct parallel poll
rscRelease or request system control
rsvRequest service and/or set the serial poll status byte
sicSend interface clear
sre Unassert or assert remote enable

cac Become active Controller

syntax **cac** 0 (take control synchronously)

cac 1 (take control immediately)

cac takes control either synchronously or immediately (and in some cases asynchronously). You generally do not need to use the **cac** function because other functions, such as **cmd** and **rpp**, take control automatically.

If you try to take control synchronously when a data handshake is in progress, the function postpones the take control action until the handshake is complete. If a handshake is not in progress, the function executes the take control action immediately. Synchronous take control is not guaranteed if a read or write operation completes with a timeout or other error.

You should take control asynchronously when it is impossible to gain control synchronously (for example, after a timeout error).

The ECIC error results if the GPIB Controller is not CIC.

cmd Send IEEE 488 commands

syntax **cmd** *string*

cmd sends GPIB command messages. These command messages include device talk and listen addresses, secondary addresses, serial and parallel poll configuration messages, and device clear and trigger messages.

You do not use **cmd** to transmit programming instructions to devices. The GPIB Read and GPIB Write functions transmit programming instructions and other device-dependent information.

string contains the command bytes the Controller sends. ASCII characters represent these bytes in **cmd** *string*. If you must send nondisplayable characters, you can enable backslash codes on the string control

or string constant or you can use a format function to list the commands in hexadecimal.

dma Set DMA mode or programmed I/O mode

syntax **dma** 0 (use programmed I/O)
dma 1 (use DMA)

dma indicates whether data transfers use DMA.

Some GPIB boards do not have DMA capability. If you try to execute **dma** 1, the function returns GPIB error 11 to indicate no capability.

gts Go from active Controller to standby

syntax **gts** 0 (no shadow handshaking)
gts 1 (shadow handshaking)

Description:

gts sets the GPIB Controller to the Controller Standby state and unasserts the ATN signal if it is the active Controller. Normally, the GPIB Controller is involved in the data transfer. **gts** permits GPIB devices to transfer data without involving the GPIB Controller.

If shadow handshaking is active, the GPIB Controller participates in the GPIB transfer as a Listener, but does not accept any data. When it detects the END message, the GPIB Controller asserts the Not Ready For Data (NRFD) to create a handshake holdoff state.

If shadow handshaking is not active, the GPIB Controller performs neither shadow handshaking nor a handshake holdoff.

If you activate the shadow handshake option, the GPIB Controller participates in a data handshake as a Listener without actually reading the data. It monitors the transfer for the END message and stops subsequent transfers. This mechanism allows the GPIB Controller to take control synchronously on subsequent operations such as **cmd** or **rpp**.

After sending the **gts** command, you should always wait for END before you initiate another GPIB command. You can do this with the GPIB Wait function.

The ECIC error results if the GPIB Controller is not CIC.

ist Set individual status bit

syntax **ist** 0 (individual status bit is cleared)
ist 1 (individual status bit is set)

ist sets the sense of the individual status (**ist**) bit.

You use **ist** when the GPIB Controller is not the CIC but participates in a parallel poll conducted by a device that is the active Controller. The CIC conducts a parallel poll by asserting the EOI and ATN signals, which send the Identify (IDY) message. While this message is active, each device that you configured to participate in the poll responds by asserting a predetermined GPIB data line either TRUE or FALSE, depending on the value of its local **ist** bit. For example, you can assign the GPIB Controller to drive the DIO3 data line TRUE if **ist** is 1 and FALSE if **ist** is 0. Conversely, you can assign it to drive DIO3 TRUE if **ist** is 0 and FALSE if **ist** is 1.

The Parallel Poll Enable (PPE) message in effect for each device determines the relationship among the

value of **ist**, the line that is driven, and the sense at which the line is driven. The GPIB Controller is capable of receiving this message either locally via **ppc** or remotely via a command from the CIC. Once the PPE message executes, **ist** changes the sense at which the GPIB Controller drives the line during the parallel poll, and the GPIB Controller can convey a one-bit, device-dependent message to the Controller.

llo Local lockout

syntax **llo**

llo places all devices in local lockout state. This action usually inhibits recognition of inputs from the front panel of the device.

llo sends the Local Lockout (LLO) command.

loc Place Controller in local state

syntax loc

loc places the GPIB Controller in a local state by sending the local message Return To Local (RTL) if it is not locked in remote mode (indicated by the LOK bit of status). You use **loc** to simulate a front panel RTL switch when you use a computer to simulate an instrument.

off Take controller offline

syntax **off**

off takes the controller offline. This is only needed when sharing the controller with another application which is using the NI 488 Library.

ppc Parallel poll configure (enable and disable)

syntax **ppc** *byte*

ppc configures the GPIB Controller to participate in a parallel poll by setting its Local Poll Enable (LPE) message to the value of *byte*. If the value of *byte* is 0, the GPIB Controller unconfigures itself.

Each of the 16 Parallel Poll Enable (PPE) messages selects the GPIB data line (DIO1 through DIO8) and sense (1 or 0) that the device must use when responding to the Identify (IDY) message during a parallel poll. The device interprets the assigned message and the current value of the individual status (ist) bit to determine if the selected line is driven TRUE or FALSE. For example, if PPE=0x64, DIO5 is driven TRUE if ist is 0 and FALSE if ist is 1. If PPE=0x68, DIO1 PPE message is in effect. You must know which **PPE** and **PPD** messages are sent and determine what the responses indicate.

ppu Parallel poll unconfigure

syntax **ppu**

ppu disables all devices from responding to parallel polls.

ppu sends the Parallel Poll Unconfigure (PPU) command.

rpp Conduct parallel poll

syntax **rpp**

rpp conducts a parallel poll of previously configured devices by asserting the ATN and EOI signals, which sends the IDY message.

rpp places the parallel poll response in the output string as ASCII characters.

rsc Release or request system control

syntax **rsc** 0 (release system control)

rsc 1 (request system control)

rsc releases or requests the capability of the GPIB Controller to send the Interface Clear (IFC) and Remote Enable (REN) messages to GPIB devices using the **sic** and **sre** functions. For the GPIB Controller to respond to IFC sent by another Controller, the GPIB Controller must not be the System Controller.

In most applications, the GPIB Controller is always the System Controller. You use **rsc** only if the computer is not the System Controller for the duration of the program execution.

rsv Request service and/or set the serial poll status byte

syntax **rsv** *byte*

rsv sets the serial poll status byte of the GPIB Controller to byte. If the 0x40 bit is set in byte, the GPIB Controller also requests service from the Controller by asserting the GPIB SRQ line. For instance, if you want to assert the GPIB SRQ line, send the ASCII character @, in which the 0x40 bit is set.

You use **rsv** to request service from the Controller using the Service Request (SRQ) signal and to provide a system-dependent status byte when the Controller serial polls the GPIB port.

sic Send interface clear

syntax **sic**

sic causes the Controller to assert the IFC signal for at least 100 msec if the Controller has System Controller authority. This action initializes the GPIB and makes the Controller port CIC. You generally use sic when you want a device to become CIC or to clear a bus fault condition.

The IFC signal resets only the GPIB functions of bus devices; it does not reset internal device functions. The Device Clear (DCL) and Selected Device Clear (SDC) commands reset the device functions. Consult the instrument documentation to determine the effect of these messages.

sre Unassert or assert remote enable

syntax **sre** 0 (unassert Remote Enable)

sre 1 (assert Remote Enable)

sre unasserts or asserts the GPIB REN line. Devices monitor REN when they select between local and remote modes of operation. A device does not actually enter remote mode until it receives its listen address.

The ESAC error occurs if the Controller is not System Controller.

GPIB Overview

Click here to access the Traditional GPIB Function Descriptions topic.

The General Purpose Interface Bus (GPIB) is a link, or interface system, through which interconnected electronic devices communicate.

GPIB History
Compatible GPIB Hardware
Traditional GPIB Functions
Traditional GPIB Function Behavior

GPIB History

Hewlett-Packard designed the GPIB (originally called the HP-IB) to interconnect and control its line of programmable instruments. The GPIB was soon applied to other applications such as intercomputer communication and peripheral control because of its 1 Mbytes/sec maximum data transfer rates. It was later accepted as IEEE Standard 488-1975 and has since evolved into ANSI/IEEE Standard 488.1-1987. The versatility of the system prompted the name General Purpose Interface Bus. For a basic description of the GPIB, see the Operation of the GPIB topic.

National Instruments brought the GPIB to users of non-Hewlett-Packard computers and devices, specializing in both high-performance, high-speed hardware interfaces and comprehensive, full-function software. The GPIB functions for LabVIEW follow the IEEE 488.1 specification.

Compatible GPIB Hardware

The following National Instruments GPIB hardware products are compatible with LabVIEW:

LabVIEW for Windows 3.x

AT-GPIB/TNT, AT-GPIB	IEEE 488.2 interface board for the IBM PC AT and compatible computers that have 16-bit plug-in slots
PCMCIA-GPIB	IEEE 488.2 interface card for computers with Type II PCMCIA slots
MC-GPIB	IEEE 488.2 interface board for the IBM PS/2 and compatible computers that have MicroChannel plug-in slots
GPIB-PCII/IIA	IEEE 488.2 interface board for the IBM PC/XT/AT
GD-GPIB	IEEE 488.2 interface board for the GRiDCASE 1500 series computer
GPIB-232CT-A	External RS-232/IEEE 488.2 Controller
GPIB-1284CT	External parallel/IEEE 488.2 Controller

LabVIEW for Windows NT

AT-GPIB/TNT, AT-GPIB	IEEE 488.2 interface board for the IBM PC AT and compatible computers that have 16-bit plug-in slots

LabVIEW for Macintosh

NB-GPIB/TNT, NB-GPIB	IEEE 488.2 interface board for Macintosh computers with full-size NuBus plug-in slots
NB-GPIB-P/TNT, NB-GPIB-P	IEEE 488.2 interface board for Macintosh computers with half-size NuBus plug-in slots
LC-GPIB	IEEE 488.2 interface board for the Macintosh LC series computers
NB-DMA2800	IEEE 488 DMA interface board for Macintosh NuBus computers
NB-DMA-8-G	IEEE 488 DMA interface board for Macintosh NuBus computers
GPIB-SCSI-A	External SCSI-to-IEEE 488.2 Controller
GPIB-SCSI	External SCSI-to-IEEE 488 Controller
GPIB-422CT	External RS-422/IEEE 488 Controller

LabVIEW for HP-UX

AT-GPIB/TNT	Plug-in ISA IEEE 488.2 interface board for the HP series 700
EISA-GPIB	Plug-in EISA IEEE 488.2 interface board for the HP series 700
GPIB-ENET	External Ethernet/IEEE 488.2 Controller for the HP series 700

LabVIEW for Sun

SB-GPIB/TNT, SB-GPIB	Plug-in IEEE 488.2 Controller for Sun SPARCstation series computers
GPIB-SCSI-A	External IEEE 488.2/SCSI Controller for Sun SPARCstation series computers
GPIB-ENET	External Ethernet/IEEE 488.2 Controller for Sun SPARCstation series computers

Traditional GPIB Functions

The traditional GPIB functions are compatible with the GPIB boards listed in the Compatible GPIB Hardware topic.

These traditional GPIB functions are compatible with existing LabVIEW programs. You can also use traditional GPIB functions with GPIB 488.2 functions to handle situations outside the scope of the IEEE 488.2 standard, such as communicating with devices that do not comply with the IEEE 488.2 standard and managing the bus in unusual ways.

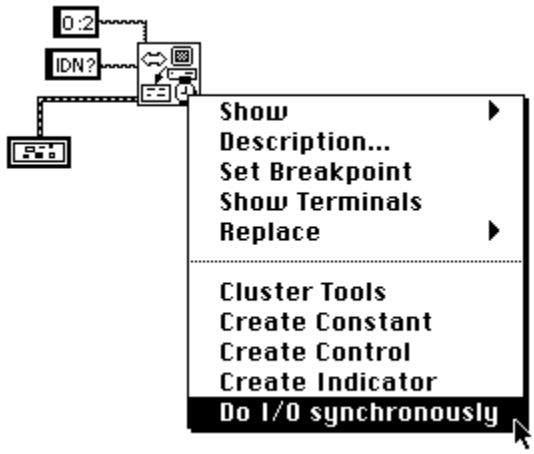
The following table lists the traditional GPIB functions:

Traditional GPIB Function	Description
GPIB Clear	Clears a single device.
GPIB Initialization	Configures characteristics of the GPIB interface.
GPIB Misc	Performs specialized GPIB operations.
GPIB Read	Reads data bytes from a GPIB device.
GPIB Serial Poll	Serial polls a single device.
GPIB Status	Indicates the status of the GPIB Controller.
GPIB Trigger	Triggers a single device.
GPIB Wait	Waits for a GPIB event to occur.
Wait for GPIB RQS	Waits for the assertion of the SRQ line Service Request.
GPIB Write	Sends data bytes to a single GPIB device.

Traditional GPIB Function Behavior

The GPIB Read and GPIB Write functions leave the device in the addressed state when they finish executing. If your device cannot tolerate being left in the addressed state, use the GPIB Misc function to send the appropriate unaddress message or configure the NI-488.2 software to unaddress automatically for all devices on the GPIB.

The traditional GPIB Read and Write functions can execute asynchronously. This means other LabVIEW activity can continue while these GPIB functions are operating. When set to execute asynchronously, a small wristwatch icon appears as part of the function icons. A popup item on the Traditional GPIB Read and GPIB Write functions allows for switching their behavior to and from asynchronous operation.


0:2
IDN?
Show
Description...
Set Breakpoint
Show Terminals
Replace
Cluster Tools
Create Constant
Create Control
Create Indicator
Do I/O synchronously

address string

address string contains the address of the GPIB device with which the function communicates. You can input both the primary and secondary addresses in **address string** by using the form primary+secondary. Both primary and secondary are decimal values, so if primary is 2 and secondary is 3, address string is 2+3.

If you do not specify an address, the functions do not perform addressing before they attempt to read and write the string. They assume you have either sent these commands another way or that another Controller is in charge and therefore responsible for the addressing. If the Controller is supposed to address the device but does not do so before the time limit expires, the functions terminate with GPIB error 6 (timeout) and set bit 14 in **status**. If the GPIB is not the Controller-In-Charge, you must not specify an address string.

When there are multiple GPIB Controllers that LabVIEW can use, a prefix to the address string in the form ID:address (or ID: if no address is necessary) determines the Controller that a specific function uses. If a Controller ID is not present, the functions assume Controller (or bus) 0.

status

status is a 16-bit Boolean array in which each bit describes a state of the GPIB Controller. If an error occurs, bit 15 is set. The error code field of the **error out** cluster is a GPIB error code only if bit 15 of **status** is set. Refer to the GPIB Status function for status bit error codes.

GPIB Clear Function

GPIB Clear

GPIB Read Function

GPIB Read

GPIB Write Function

GPIB Write

Wait for GPIB RQS Function

Wait for GPIB RQS Function

GPIB Trigger Function

GPIB Trigger

GPIB Serial Poll Function

GPIB Serial Poll

GPIB Initialization Function

GPIB Initialization

GPIB Status Function

GPIB Status

GPIB Wait Function

GPIB Wait

GPIB Misc Function

GPIB Misc

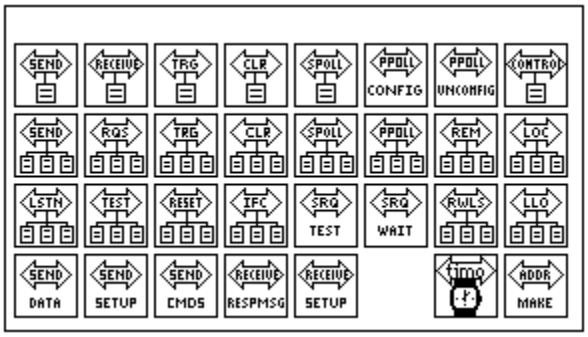
GPIB 488.2 Function Descriptions

Click here to access the GPIB 488.2 Overview topic.

This topic describes the IEEE 488.2 (GPIB) Functions. GPIB functions are divided into five groups--Single-Device Functions, Multiple-Device Functions, Bus Management Functions, Low-Level I/O Functions, and General Functions .

The following figure shows the **GPIB 488.2** palette which you access by selecting **Functions**»**Instrument I/O**»**GPIB 488.2**:

Click on one of the icons below for VI description information. You can also click on one of the text jumps below the icons to access VI descriptions.



AllSPoll
DevClear
DevClearList
EnableLocal
EnableRemote
FindLstn
FindRQS
MakeAddr
PassControl
PPoll
PPollConfig
RcvRespMsg
ReadStatus
Receive
ReceiveSetup
ResetSys
Send
SendCmds
SendIFC
SendList
SendLLO
SendDataBytes
SendSetup
SetRWLS
SetTimeOut
TestSRQ
TestSys
Trigger
TriggerList
WaitSRQ

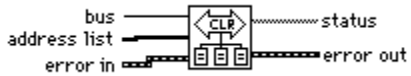
For examples of how to use the GPIB 488.2 Functions, see `examples\instr\smplgpib.llb`.

Single-Device Functions

DevClear
PassControl
PPollConfig
ReadStatus
Receive
Send
Trigger

DevClear

Clears a single device. To send the Selected Device Clear (SDC) message to several GPIB devices, use the DevClearList function.



bus. See the bus topic for more information.

address. The function sends the GPIB SDC message to the device at **address**. If the **address** terminal is not wired, the function sends the Universal Device Clear message to all devices on the GPIB. See the address topic for more information.

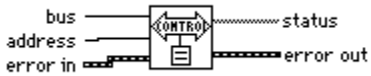
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

PassControl

Passes control to another device with Controller capability.



bus. See the bus topic for more information.

address. The function sends the GPIB Device Take Control message to the device specified by address. See the address topic for more information.

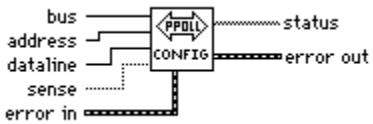
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

PPollConfig

Configures a device for parallel polls.



bus. See the bus topic for more information.

address. The function configures the GPIB device at address for parallel polls according to the dataline and sense parameters. See the address topic for more information.

dataline is the data line (1 to 8) on which the device responds to the parallel poll.

sense indicates the condition under which the data line is asserted or unasserted. The device compares this sense value (TRUE or FALSE) to its individual status bit and responds accordingly.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

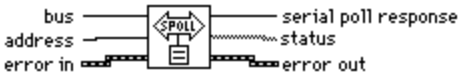
status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

Devices have the option of configuring themselves for parallel polls, in which case they ignore attempts by the Controller to configure them. You should determine whether the device is locally or remotely configured before you use PPollConfig or PPollUnconfig.

ReadStatus

Serial polls a single device to get its status byte.



bus. See the bus topic for more information.

address. The function serial polls the device at **address**. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

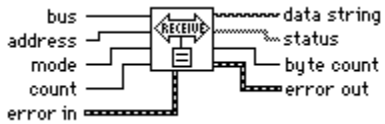
serial poll response is the status byte of the device at **address**. You can learn how to interpret this byte from the documentation supplied with your device.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

Receive

Reads data bytes from a GPIB device.



Receive terminates when the function does one of the following:

- reads the number of bytes requested
- detects an error
- exceeds the time limit

- detects the END message (EOI asserted)
- detects the EOS character (assuming the value supplied to **mode** has enabled this option)

bus. The function addresses the interface indicated by bus as a Listener. See the bus topic for more information.

address. The function addresses the device indicated by address as a Talker. See the address topic for more information.

mode selects the method that signals the end of the data. If **mode** is a value from decimal 0 through 255, the ASCII character that corresponds to it is the termination character, and the function stops the read when it detects the character. If **mode** is not wired, or is decimal 256, the function stops the read when it detects END.

count. The function returns up to count data bytes from the device in data string.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

data string contains **count** data bytes from the GPIB device.

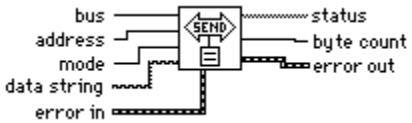
status. See the status topic for more information.

byte count. See the byte count topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

Send

Sends data bytes to a single GPIB device.



bus. The function addresses the interface indicated by bus as a Talker. See the bus topic for more information.

address. The function addresses the device indicated by **address** as a Listener. If you do not wire **address**, the function sends data string with no addressing, thereby sending it to all previously addressed Listeners. To send **data string** to several devices, use the SendList function. See the address topic for more information.

mode describes how to signal the end of the data to the Listener.

0: Do nothing to mark the end of the transfer.
1: Send NL (linefeed) with EOI after the data bytes.
2: Send EOI with the last data byte in the string.

data string contains the data bytes that the function sends to the GPIB device.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

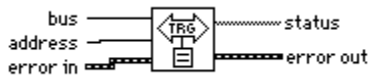
status. See the status topic for more information.

byte count. See the byte count topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

Trigger

Triggers a single device. To send a single message that triggers several GPIB devices, use the TriggerList function.



bus. See the bus topic for more information.

address. The function sends the GPIB Group Execute Trigger message to the device at **address.** If the **address** terminal is not wired, the function sends the Group Execute Trigger message with no addressing, and this triggers all previously addressed Listeners. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

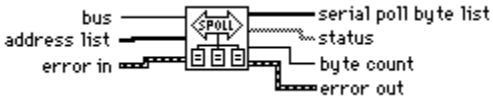
Multiple-Device Functions

AllSPoll
DevClearList
EnableLocal
EnableRemote
FindRQS
PPoll
PPollUnconfig
SendList
TriggerList

AllSPoll

Serial polls all devices.

Although the AllSPoll function is general enough to serial poll any number of GPIB devices, you should use the ReadStatus function when you serial poll only one GPIB device.



bus. See the bus topic for more information.

address list. The function serial polls GPIB devices whose addresses appear in **address list**. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

serial poll byte list contains the responses of the devices whose addresses are contained in the corresponding elements of **address list**.

status. See the status topic for more information.

byte count contains the index of any device that times out instead of responding to the poll. See the byte count topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

DevClearList

Clears multiple devices simultaneously.

bus. See the bus topic for more information.

address list. The function sends the GPIB Selected Device Clear (SDC) message to the devices listed in **address list**. If the **address list** terminal is not wired, the function sends the Universal Device Clear message to all devices on the GPIB. See the address topic for more information.

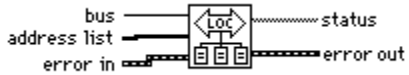
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

EnableLocal

Enables local mode for multiple devices.



bus. See the bus topic for more information.

address list. The function sends the GPIB Selected Device Clear (SDC) message to the devices listed in **address list**. If the **address list** terminal is not wired, the function sends the Universal Device Clear message to all devices on the GPIB. See the address topic for more information.

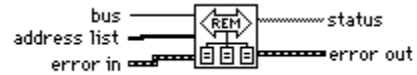
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

EnableRemote

Enables remote programming of multiple GPIB devices.



bus. See the bus topic for more information.

address list. The function sends the GPIB Selected Device Clear (SDC) message to the devices listed in **address list**. If the **address list** terminal is not wired, the function sends the Universal Device Clear message to all devices on the GPIB. See the address topic for more information.

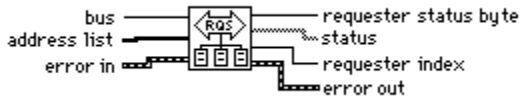
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

FindRQS

Determines which device is requesting service.



bus. See the bus topic for more information.

address list. The function sends the GPIB Selected Device Clear (SDC) message to the devices listed in **address list**. If the **address list** terminal is not wired, the function sends the Universal Device Clear message to all devices on the GPIB. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

requester status byte is the status byte for the device asserting SRQ.

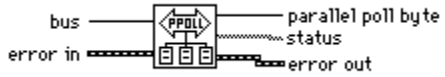
status. See the status topic for more information.

requester index is the index of the address in **address list** of the device asserting SRQ. Arrays are zero-indexed. A requester index of -1 indicates that none of the devices specified in the **address list** are requesting service.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

PPoll

Performs a parallel poll.



bus. See the bus topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

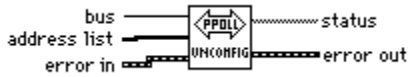
parallel poll byte is the result of the parallel poll. Each bit of the poll result returns one bit of **status** information from each device configured for parallel polls. The state of each bit (0 or 1) and the interpretation of these states are based on the latest parallel poll configuration sent to the devices and on the individual status of the devices.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

PPollUnconfig

Unconfigures devices for parallel polls. The function unconfigures the GPIB devices whose addresses are contained in the **address list** array for parallel polls; that is, they no longer participate in polls.



bus. See the bus topic for more information.

address list. If the **address list** terminal is not wired, the function sends the GPIB Parallel Poll Unconfigure (PPU) message, unconfiguring all devices. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

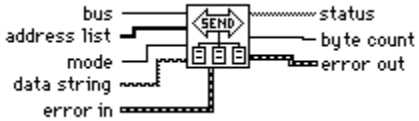
status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error

In/Error Out for error information specific to GPIB.

SendList

Sends data bytes to multiple GPIB devices. This function is similar to Send, except that SendList sends data to multiple Listeners with only one transmission.



bus. See the bus topic for more information.

address list. The **address list** array contains a list of GPIB addresses for devices, which the function addresses as Listeners See the address topic for more information.

mode describes how to signal the end of the data to the Listener.

0: Do nothing to mark the end of the transfer.
1: Send NL (linefeed) with EOI after the data bytes.
2: Send EOI with the last data byte in the string.

data string contains the data bytes that the function sends to the addressed devices.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

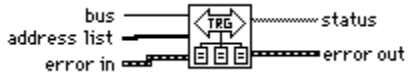
status. See the status topic for more information.

byte count. See the byte count topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

TriggerList

Triggers multiple devices simultaneously.



bus. See the bus topic for more information.

address list. The function simultaneously triggers GPIB devices whose addresses appear in **address list**. If the **address list** terminal is not wired, the function sends the Group Execute Trigger message without addressing, which triggers all previously addressed Listeners. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

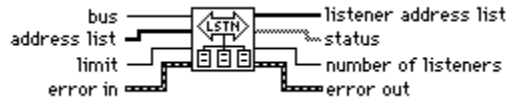
Bus Management Functions

FindLstn
ResetSys
SendIFC
SendLLO
SetRWLS

TestSRQ
TestSys
WaitSRQ

FindLstn

Finds all Listeners on the GPIB. You normally use this function to detect the presence of devices at particular addresses because most GPIB devices have the ability to listen. When you detect them, you can usually interrogate the devices with to determine their identity.



bus. See the bus topic for more information.

address list. The function tests for the presence of a Listener for all devices listed in **address list**. If the function does not detect a listening device at the particular primary address, it tests all the secondary addresses associated with that primary address. See the address topic for more information.

limit. If more Listeners are present on the bus than **limit** specifies, the function truncates listener **address list** after **limit** entries.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

listener address list contains the addresses of all the Listeners the function finds.

status. See the status topic for more information.

number of listeners contains the number of addresses placed in **listener address list**.

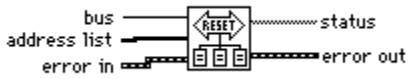
error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

ResetSys

Performs bus initialization, message exchange initialization, and device initialization. First, the function asserts Remote Enable (REN), followed by Interface Clear (IFC), unaddressing all devices and making the GPIB board (the System Controller) the Controller-in-Charge.

Second, the function sends the Device Clear (DCL) message to all connected devices. This ensures that all IEEE 488.2-compatible devices can receive the Reset (RST) message that follows.

Third, the function sends the *RST message to all devices whose addresses are contained in the address list array. This message initializes device-specific functions within each device.



bus. See the bus topic for more information.

address list. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

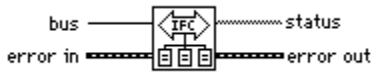
status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

SendIFC

Clears the GPIB functions with Interface Clear (IFC). When you issue the GPIB Device IFC message, the interface functions of all connected devices return to their cleared states.

You should use this function as part of a GPIB initialization. It forces the GPIB board to be Controller of the GPIB and ensures that the connected devices are all unaddressed and that the interface functions of the devices are in their idle states.



bus. See the bus topic for more information.

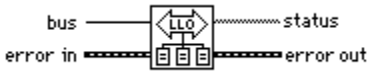
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

SendLLO

Sends the Local Lockout (LLO) message to all devices. When the function sends the GPIB Local Lockout message, a device cannot independently choose the local or remote state. While Local Lockout is in effect, only the Controller can alter the local or remote state of the devices by sending the appropriate GPIB messages.You should use SendLLO only in unusual local/remote situations, particularly those in which you must lock all devices into local programming state. Use the SetRWLS Function when you want to place devices in Remote Mode With Lockout State.



bus. See the bus topic for more information.

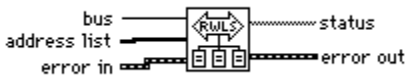
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

SetRWLS

Places particular devices in the Remote With Lockout State. The function sends Remote Enable (REN) to the GPIB devices listed in **address list**. It also places all devices in Lockout State, which prevents them from independently returning to local programming mode without intervention by the Controller.



bus. See the bus topic for more information.

address list. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

TestSRQ

Determines the current state of the SRQ line. This function is similar in format to the WaitSRQ function, except that WaitSRQ suspends itself while it waits for an occurrence of SRQ, and TestSRQ immediately returns the current SRQ state.

bus. See the bus topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

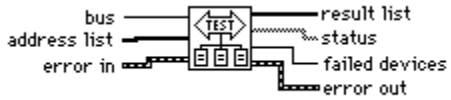
SRQ is TRUE if the **SRQ** line is asserted and FALSE if it is not.

status. See the status topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

TestSys

Directs multiple devices to conduct IEEE 488.2 self-tests.



bus. See the bus topic for more information.

address list. The function simultaneously sends to the GPIB devices whose addresses appear in **address list** a message that instructs them to conduct their self-test procedures. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

result list. Each device returns an integer code signifying the results of its tests, and the function returns these codes in the corresponding elements of result list. A result code of 0 indicates that the device passed its tests, and any other value indicates that the tests resulted in an error.

status. See the status topic for more information.

failed devices contains the number of devices that failed their tests.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

WaitSRQ

Waits until a device asserts Service Request. The function suspends execution until a GPIB device connected on the GPIB asserts the Service Request (SRQ) line.

This function is similar in format to TestSRQ, except that TestSRQ returns the SRQ status immediately, whereas WaitSRQ suspends the program for the duration of the timeout period (but no longer) waiting for an SRQ to occur.

bus. See the bus topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error

In/Error Out for error information specific to GPIB.

SRQ. If the **SRQ** occurs within the timeout period specified in the GPIB configuration, the function returns TRUE in **SRQ**. Otherwise, the function returns FALSE.

status. See the status topic for more information.

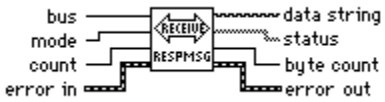
error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

Low-Level I/O Functions

RcvRespMsg
ReceiveSetup
SendCmds
SendDataBytes
SendSetup

RcvRespMsg

Reads data bytes from a previously addressed device. This function assumes that another function, such as ReceiveSetup, Receive, or SendCmds, has already addressed the GPIB Talkers and Listeners. You use RcvRespMsg specifically to skip the addressing step of GPIB management. You normally use the Receive function to perform the entire sequence of addressing and then to receive the data bytes.



bus. See the bus topic for more information.

mode determines the data termination method. If **mode** is from 0 through decimal 255, the ASCII character that corresponds to it is the termination character, and the function stops the read when it detects the character. If **mode** is not wired or is decimal 256, the read stops when the function detects the END message (EOI).

count. The function reads up to **count** data bytes from the GPIB.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

data string contains the bytes read from the GPIB.

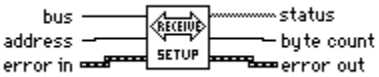
status. See the status topic for more information.

byte count. See the byte count topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

ReceiveSetup

Prepares a device to send data bytes and prepares the GPIB board to read data bytes. After you call this function, you can use a function such as RcvRespMsg to transfer the data from the Talker. In this way, you eliminate the need to re-address the devices between blocks of reads.



bus. The function addresses the interface indicated by bus as a Listener. See the bus topic for more information.

address. The function addresses the device specified by **address** as a Talker. See the address

topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

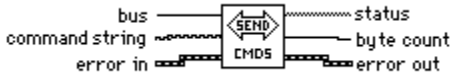
byte count. See the byte count topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

SendCmds

Sends GPIB command bytes.

You normally do not need to use SendCmds for GPIB operation. You use it when specialized command sequences, not provided for in other functions, must be sent over the GPIB.



bus. See the bus topic for more information.

command string contains the command bytes the function sends over the GPIB.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

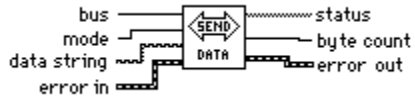
status. See the status topic for more information.

byte count. See the byte count topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

SendDataBytes

Sends data bytes to previously addressed devices.



bus. See the bus topic for more information.

mode indicates the method of signaling the end of the data to the Listeners.

0: Do nothing to mark the end of the transfer.
1: Send NL (linefeed) with EOI after the data bytes.
2: Send EOI with the last data byte in the string.

data string contains the data bytes the function sends over the GPIB.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

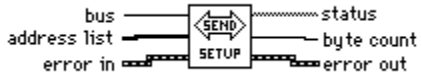
byte count. See the byte count topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

SendSetup

Prepares particular devices to receive data bytes. You normally follow a call to this function with a call to a function such as SendDataBytes to actually transfer the data to the Listeners. This sequence eliminates

the need to re-address the devices between blocks of sends.



bus. See the bus topic for more information.

address list. The function addresses GPIB devices whose addresses appear in **address list** as Listeners. See the address topic for more information.

error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

status. See the status topic for more information.

byte count. See the byte count topic for more information.

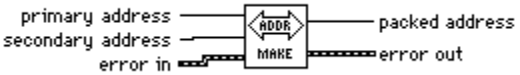
error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

General Functions

MakeAddr
SetTimeOut

MakeAddr

Combines primary address and secondary address in a specially formatted packed address for devices that require both a primary and secondary GPIB address.



primary address. The function places **primary address** in the lower byte of the packed address.

secondary address. The function places **secondary address** in the upper byte of the packed address.

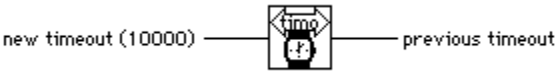
error in. See the Error Reporting topic for a detailed description of **error in**. See GPIB Error In/Error Out for error information specific to GPIB.

packed address contains primary address in its lower byte and secondary address in its upper byte. You can use **packed address** as the address input to the GPIB 488.2 Functions.

error out. See the Error Reporting topic for a detailed description of **error out**. See GPIB Error In/Error Out for error information specific to GPIB.

SetTimeOut

Changes the global timeout period for all GPIB 488.2 Functions. This function also sets the default timeout period for all GPIB functions.



new timeout defaults to 10,000 ms.

previous timeout is the **previous timeout** period for all GPIB 448.2 Functions.

GPIB 488.2 Overview

Click here to access the GPIB 488.2 Function Descriptions topic.

IEEE 488.2 Standard
LabVIEW GPIB 488.2 Functions

IEEE 488.2 Standard

The ANSI/IEEE Standard 488.2-1987 expanded on the earlier IEEE 488.1 standard to describe exactly how the Controller should manage the GPIB, including the standard messages that compliant devices should understand, the mechanisms for reporting device errors and other status information, and the various protocols that discover and configure compliant devices connected to the bus.

The original standard, renamed IEEE 488.1, addressed only the hardware specifications of the GPIB cable and basic protocols. Its main shortcoming was that it left the interpretation of the standard as it applied to GPIB devices up to the instrument manufacturers. Thus, each GPIB instrument had a unique command set. To integrate each instrument into a particular GPIB system, programmers had to learn programming particulars for each device, a time-consuming and frustrating process. IEEE 488.2 specifically states how compliant devices must communicate. This standard, along with Standard Commands for Programmable Instruments (SCPI), which defines specific function-dependent command sets, makes instrument programming more uniform.

The IEEE 488.2 standard also addresses Controller issues, such as the capabilities a compatible Controller must have. For example, the ability to monitor any of the bus lines at any time is crucial for detecting active devices (Talkers and Listeners) on the GPIB. IEEE 488.2 also defines the bus commands and protocols a Controller must use. The new standard also lists minimum functionality requirements, which directly influence the style of the NI-488.2 software in general and the GPIB 488.2 functions for LabVIEW in particular. See the Operation of the GPIB, topic for more information on Talkers, Listeners, and Controllers.

LabVIEW GPIB 488.2 Functions

Using LabVIEW GPIB 488.2 functions together with IEEE 488.2-compatible devices improves the predictability of instrument and software behavior and lessens programming differences between instruments of different manufacturers.

The latest revisions of many National Instruments GPIB boards are fully compatible with the IEEE 488.2 specification for Controllers. The LabVIEW package also contains functions that make use of IEEE 488.2. By using these functions, your programming interface will strictly adhere to the IEEE 488.2 standard for command and data sequences.

Single-Device Functions
Multiple-Device Functions
Bus Management Functions
Low-Level Functions
General Functions

The GPIB 488.2 functions contain the same basic functionality as the traditional GPIB functions, and include the following enhancements and additions:

- You specify the GPIB device address with an integer instead of a string. Further, you specify the bus number with an additional numeric control, which makes dealing with multiple GPIB interfaces easier.
- You can determine the GPIB status, error, and/or byte count immediately from the connector pane of each GPIB 488.2 function. You no longer need to use the GPIB Status Function to obtain error and other information.

- The FindLstn Function implements the IEEE 488.2 Find All Listeners protocol. You can use this function at the beginning of an application to determine which devices are present on the bus without knowing their addresses.
- The GPIB Misc Function is still available, but it is no longer necessary in most cases. IEEE 488.2 specifies routines for most GPIB application needs, which are implemented as functions. However, you can mix the GPIB Misc Function, as well as other GPIB functions, with the GPIB 488.2 functions if you need to.
- There are GPIB 488.2 functions with low-level as well as high-level functionality, to suit any GPIB application. You can use the low-level functions in Non-Controller situations or when you need additional flexibility.
- Although you must use an IEEE 488.2-compatible Controller to use these functions, they can control both IEEE 488.1 and IEEE 488.2 devices. The GPIB 488.2 functions are divided into five functional categories: single-device, multiple-device, bus management, low-level, and general.

Single-Device Functions

The single-device functions perform GPIB I/O and control operations with a single GPIB device. In general, each function accepts a single-device address as one of its inputs. The following table lists the single-device functions:

Single-Device Function	Description
DevClear	Clears a single device.
PPollConfig	Configures a device for parallel polls.
PassControl	Passes control to another device with Controller capability.
ReadStatus	Serial polls a single device to get its status byte.
Receive	Reads data bytes from a GPIB device.
Send	Sends data bytes to a single GPIB device.
Trigger	Triggers a single device.

Multiple-Device Functions

The multiple-device functions perform GPIB I/O and control operations with several GPIB devices at once. In general, each function accepts an array of addresses as one of its inputs.

The following table lists the mutiple-device functions:

Multiple-Device Function	Description
AllSpoll	Serial polls all devices.
DevClearList	Clears multiple devices simultaneously.
EnableLocal	Enables operations from the devices' physical front

	panels.
EnableRemote	Enables remote GPIB programming of devices.
FindRQS	Determines which device is requesting service.
PPoll	Performs a parallel poll.
PPollUnconfig	Unconfigures a device for parallel polls.
SendList	Sends data bytes to multiple GPIB devices.
TriggerList	Triggers multiple devices simultaneously.

Bus Management Functions

The bus management functions perform system-wide functions or report system-wide status. The following table lists the bus management functions:

Bus Management Function	Description
FindLstn	Finds all Listeners on the GPIB.
ResetSys	Performs an IEEE 488.2 three-step system initialization.
SendIFC	Clears the GPIB functions with Interface Clear.
SendLLO	Sends the Local Lockout message to all devices.
SetRWLS	Places devices in the Remote Mode With Lockout State.
TestSRQ	Determines the state of the SRQ line.
TestSys	Instructs IEEE 488.2 devices to conduct self-tests.
WaitSRQ	Suspends operation until SRQ is asserted.

Low-Level Functions

The low-level functions let you create a more specific, detailed program than higher-level functions. You use low-level functions for unusual situations or for situations requiring additional flexibility.

The following table lists the low-level functions:

Low-Level Function	Description
SendDataBytes	Sends data bytes to previously addressed devices.

RcvRespMsg	Reads data bytes from a previously addressed device.
ReceiveSetup	Prepares a device to send data bytes and prepares the GPIB interface to read them.
SendCmds	Sends GPIB command bytes.
SendSetup	Prepares particular devices to receive data bytes.

General Functions

The general functions are useful for special situations. The following table lists the general functions:

General Function	Description
MakeAddr	Formats a primary and secondary address for use with the GPIB 488.2 Functions.
SetTimeOut	Changes the global timeout period for all GPIB 488.2 Functions and sets the default timeout period for all GPIB functions.

address

address contains the primary address of the GPIB device with which the function communicates. If a secondary address is required, use the MakeAddr function to put the primary and secondary addresses in the proper format. Unless specified otherwise, **address** and **address list** are data types integer and integer array, respectively.

The default primary address of the GPIB board is 0, with no secondary address. It is designated as System Controller. The default timeout value for the functions is 10 seconds. If you want to change any of these parameters, use the configuration utility included with your GPIB board. You can also use the GPIB Init and SetTimeOut functions to set the primary address and to change the default timeout value at run time, but these functions affect the interface only when you use it with LabVIEW. For more information, see the documentation supplied with your hardware interface.

bus

bus refers to the GPIB bus number. If you have only one GPIB interface in your computer, the default bus number is 0. For additional GPIB interfaces, see the software installation instructions included with your GPIB board.

byte count

byte count refers to the number of bytes that pass over the GPIB.

status

status is a Boolean array in which each bit describes a state of the GPIB Controller. If an error occurs, the GPIB functions set bit 15. **GPIB error** is valid only if bit 15 of **status** is set. See the status bit and GPIB error tables in the GPIB Status function description topic for more information.

Send Function

Send

Receive Function

Receive

Trigger Function

Trigger

DEVClear Function

DevClear

ReadStatus Function

ReadStatus

PPollConfig Function

PPollConfig

PPollUnconfig Function

PPollUnconfig

PassControl Function

PassControl

SendList Function

SendList

FindRQS Function

FindRQS

TriggerList Function

TriggerList Function

DevClearList Function

DevClearList

AllSpoll Function

AllSPoll

PPoll Function

PPoll

EnableRemote Function

EnableRemote

EnableLocal Function

EnableLocal

FindLstn Function

FindLstn

TestSys Function

TestSys

ResetSys Function

ResetSys

SendIFC Function

SendIFC

TestSRQ Function

TestSRQ

WaitSRQ Function

WaitSRQ

SetRWLS Function

SetRWLS

SendLLO Function

SendLLO

SendDataBytes Function

SendDataBytes

SendSetup Function

SendSetup

SendCmds Function

SendCmds

RcvRespMsg Function

RcvRespMsg

ReceiveSetup Function

ReceiveSetup

SetTimeOut Function

SetTimeOut

MakeAddr Function

MakeAddr

Serial Port VI Descriptions

Click here to access the Serial Port VI Overview topic.

This topic describes the VIs for serial port operations.

The following figure shows the **Serial** palette which you access by selecting **Functions**»**Instrument I/O**»**Serial**:

Click on one of the icons below for VI description information. You can also click on one of the text jumps below the icons to access VI descriptions.

Bytes at Serial Port
Serial Port Break
Serial Port Init
Serial Port Read
Serial Port Write

For examples of how to use the Serial Port VIs, see `examples\instr\smplserl.llb`.

Bytes at Serial Port

Returns in **byte count** the number of bytes in the input buffer of the serial port indicated in **port number**.

port number. See the Port Number topic for a list of valid port numbers.

byte count is the number of bytes currently queued up in the serial port buffer.

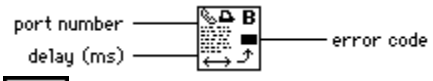
error code. If **error code** is non-zero, an error occurred. Refer to the Error Code topic, for a list of error codes.

You can connect **error code** to one of the error handler VIs. These VIs describe the error and give you options on how to proceed when an error occurs. For more information on using the error handler VIs, refer to Error Handler VIs.

Some error codes returned by the serial port VIs are platform-specific. Please refer to your system documentation for a list of error codes.

Serial Port Break

Sends a break on the output port specified by **port number** for a period of time at least as long as the **delay** input requests.



port number. See the Port Number topic for a list of valid port numbers.

delay is the time interval in msec for which the break must last. The actual period of the break can be longer than this interval. On the Sun, **delay** is ignored; the break occurs for at least 0.25 sec and no more than 0.5 sec.

error code. If **error code** is non-zero, an error occurred. Refer to the Error Code topic, for a list

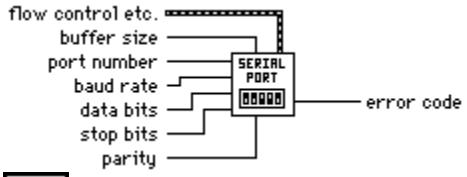
of error codes.

You can connect **error code** to one of the error handler VIs. These VIs describe the error and give you options on how to proceed when an error occurs For more information on using the error handler VIs, refer to Error Handler VIs.

Some error codes returned by the serial port VIs are platform-specific. Please refer to your system documentation for a list of error codes.

Serial Port Init

Initializes the selected serial port to the specified settings.



flow control etc. contains the following parameters.

Input XON/XOFF. See the Handshaking Modes topic for more information.

Input HW Handshake. On the PC and SPARCstation, this parameter corresponds to Request To Send (RTS) handshaking.

Input alt HW Handshake. On the PC, this parameter corresponds to Data Terminal Ready (DTR) handshaking. On the SPARCstation, this parameter is ignored.

Output XON/XOFF. See the Handshaking Modes topic for more information.

Output HW Handshake. On the PC and SPARCstation, this parameter corresponds to Clear to Send (CTS) handshaking.

Output alt HW Handshake. On the PC, this parameter corresponds to Data Set Ready (DSR) handshaking. On the SPARCstation, this parameter is ignored.

XOFF byte is the byte used for XOFF (^S).

XON byte is the byte used for XON (^Q).

Parity Error Byte. If the high byte is non-zero, the low byte is the character that is used to replace any parity errors found when **parity** is enabled.

buffer size indicates the size of the input and output buffers the VI allocates for communication through the specified port. If **buffer size** is less than or equal to 1 K, the VI uses 1 K as the buffer size. You may need to use larger buffers for large data transfers. The **buffer size** is in bytes.

port number. See the Port Number topic for a list of valid port numbers.

baud rate is the rate of transmission.

data bits is the number of bits in the incoming data. The value of **data bits** is between five and eight.

stop bits is 0 for one stop bit, 1 for one-and-a-half **stop bits**, or 2 for two **stop bits**.

parity is 0 for no parity, 1 for odd parity, 2 for even parity, 3 for mark parity, or 4 for space parity.

error code is -1 if **baud rate**, **data bits**, **stop bits**, **parity**, or **port number** are out of range, or if the serial port could not be initialized. Check the values of **baud rate**, **data bits**, **stop bits**, **parity**, and **port number**. If these values are valid, verify that the serial port has been initialized. Refer to the Error Code topic, for a list of error codes.

You can connect **error code** to one of the error handler VIs. These VIs can describe the error and give you options on how to proceed when an error occurs. . For more information on using the error handler VIs, refer to Error Handler VIs.

Some error codes returned by the serial port VIs are platform-specific. Please refer to your system documentation for a list of error codes.

Serial Port Read

Reads the number of characters specified by requested **byte count** from the serial port indicated in **port number**.

port number. See the Port Number topic for a list of valid port numbers.

requested byte count specifies the number of characters to be read. If you want to read all of the characters currently at the serial port, first execute the Bytes at Serial Port VI to determine the exact number of bytes ready to be read. Then use the **byte count** output of that VI as the **requested byte count** input to the Serial Port Read VI.

string read. The VI returns the bytes read in **string read**.

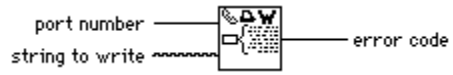
error code. If **error code** is non-zero, an error occurred Refer to the Error Code topic, for a list of error codes.

You can connect **error code** to one of the error handler VIs. These VIs describe the error and give you options on how to proceed when an error occurs. For more information on using the error handler VIs, refer to Error Handler VIs.

Some error codes returned by the serial port VIs are platform-specific. Please refer to your system documentation for a list of error codes.

Serial Port Write

Writes the data in string to write to the serial port indicated in port number.



port number. See the Port Number topic for a list of valid port numbers.

string to write is the data to be written to the serial port.

error code. If **error code** is non-zero, an error occurred. Refer to the Error Code topic, for a list of error codes.

You can connect **error code** to one of the error handler VIs. These VIs describe the error and give you options on how to proceed when an error occurs. For more information on using the error handler VIs, refer to Error Handler VIs.

Some error codes returned by the serial port VIs are platform-specific. Please refer to your system documentation for a list of error codes.

Serial Port VI Overview

Click here to access the Serial Port VI Descriptions topic.

Serial Port VIs

Serial Port VIs

The serial port VIs configure the serial port of your computer and conduct I/O using that port. The following table lists the serial port VIs:

Serial Port VI	Description
Bytes at Serial Port	Reports the number of bytes available in the serial port input buffer.
Serial Port Break	Sends a break on the specified output port.
Serial Port Init	Initializes the selected serial port by specifying the protocol, handshaking, and buffer size.
Serial Port Read	Reads characters from the specified serial port.
Serial Port Write	Writes a data string to the specified serial port.

Port Number

When you use the serial port VIs under Windows 3.x, the **port number** parameter has the following values:

0:	COM1	5:	COM6	10:	LPT1
1:	COM2	6:	COM7	11:	LPT2
2:	COM3	7:	COM8	12:	LPT3
3:	COM4	8:	COM9	13:	LPT4
4:	COM5				

When you use the serial port VIs under Windows 95 or Windows NT, the **port number** parameter is 0 for COM1, 1 for COM2, and so on.

On the Macintosh, port 0 is the modem, using the drivers `.ain` and `.aout`. Port 1 is the printer, using the drivers `.bin` and `.bout`. To get more ports on a Macintosh, you must install other boards, with the accompanying drivers.

On a Sun SPARCstation under Solaris 1, the port number parameter for the serial port VIs is 0 for `/dev/ttya`, 1 for `/dev/ttyb`, and so on. Under Solaris 2, port 0 refers to `/dev/cua/a`, 1 to `/dev/cua/b`, and so on. Under HP-UX port number 0 refers to `/dev/tty0p0`, 1 to `/dev/tty1p0`, and so on.

For more information about configuring your serial ports, see the common questions for Serial I/O topic.

Handshaking Modes

A common problem in serial communications is ensuring that both sender and receiver keep up with data transmission. The serial port driver can buffer incoming/outgoing information, but that buffer is of a finite size. When it becomes full, the computer ignores new data until you have read enough data out of the buffer to make room for new information.

Handshaking helps prevent this buffer from overflowing. With handshaking, the sender and the receiver notify each other when their buffers fill up. The sender can then stop sending new information until the other end of the serial communication is ready for new data.

You can perform two kinds of handshaking in LabVIEW--software handshaking and hardware handshaking. You can turn both of these forms of handshaking on or off using the Serial Port Init VI. By default, the VIs do not use handshaking.

Software Handshaking–XON/XOFF

XON/XOFF is a software handshaking protocol you can use to avoid overflowing serial port buffers. When the receive buffer is nearly full, the receiver sends XOFF (<control-S> [decimal 19]) to tell the other device to stop sending data. When the receive buffer is sufficiently empty, the receiver sends XON (<control-Q> [decimal 17]) to indicate that transmission can begin again. When you enable XON/XOFF, the devices always interpret <control-Q> and <control-S> as XON and XOFF characters, never as data. When you disable XON/XOFF, you can send <control-Q> and <control-S> as data. Do not use XON/XOFF with binary data transfers because <control-Q> or <control-S> may be embedded in the data, and the devices will interpret them as XON and XOFF instead of as data.

Error Codes

You can connect the **error code** parameter to one of the error handler VIs. These VIs can describe the error and give you options on how to proceed when an error occurs. . For more information on using error handling, refer to the Error Handling topic.

Some error codes returned by the serial port VIs are platform-specific. Please refer to your system documentation for a list of error codes. For LabVIEW Serial I/O Error codes, refer to the Serial Port Error Codes topic.

Bytes at Serial Port VI

Bytes at Serial Port

Serial Port Break VI

Serial Port Break

Serial Port Init VI

Serial Port Init

Serial Port Read VI

Serial Port Read

Serial Port Write VI

Serial Port Write

VISA Error Codes

Code	Name	Description
-1073807360	VI_ERROR_SYSTEM_ERROR	Unknown system error (miscellaneous error).
-1073807346	VI_ERROR_INV_OBJECT VI_ERROR_INV_SESSION	The given session or object reference is invalid.
-1073807344	VI_ERROR_INV_EXPR	Invalid expression specified for search.
-1073807343	VI_ERROR_RSRC_NFOUND	Insufficient location information or the device or resource is not present in the system.
-1073807342	VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.
-1073807341	VI_ERROR_INV_ACC_MODE	Invalid access mode.
-1073807339	VI_ERROR_TMO	Timeout expired before operation completed.
-1073807338	VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.
-1073807331	VI_ERROR_NSUP_ATTR	The specified attribute is not defined or supported by the referenced resource.
-1073807330	VI_ERROR_NSUP_ATTR_STATE	The specified state of the attribute is not valid, or is not supported as defined by the resource.
-1073807329	VI_ERROR_ATTR_READONLY	The specified attribute is read-only.
-1073807322	VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
-1073807321	VI_ERROR_INV_MECH	Invalid mechanism specified.
-1073807320	VI_ERROR_HNDLR_NINSTALLED	A handler was not installed.
-1073807319	VI_ERROR_INV_HNDLR_REF	The given handler reference is invalid.
-1073807318	VI_ERROR_INV_CONTEXT	Specified event context is invalid.
-1073807308	VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol

		occurred during transfer.
-1073807307	VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
-1073807306	VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error during transfer.
-1073807305	VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error during transfer.
-1073807304	VI_ERROR_BERR	Bus error occurred during transfer.
-1073807302	VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
-1073807300	VI_ERROR_ALLOC	Insufficient system resources to perform necessary memory allocation.
-1073807299	VI_ERROR_INV_MASK	Invalid buffer mask specified.
-1073807298	VI_ERROR_IO	Could not perform read/write operation because of I/O error.
-1073807297	VI_ERROR_INV_FMT	A format specifier in the format string is invalid.
-1073807295	VI_ERROR_NSUP_FMT	A format specifier in the format string is not supported.
-1073807294	VI_ERROR_LINE_IN_USE	The specified trigger line is currently in use.
-1073807286	VI_ERROR_SRQ_NOCCURRED	Service request has not been received for the session.
-1073807282	VI_ERROR_INV_SPACE	Invalid address space specified.
-1073807279	VI_ERROR_INV_OFFSET	Invalid offset specified.
-1073807276	VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
-1073807273	VI_ERROR_WINDOW_NMAPPED	The specified session is not currently mapped.
-1073807265	VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).

-1073807264	VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.
-1073807257	VI_ERROR_NSUP_OPER	The given session or object reference does not support this operation.
-1073807242	VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
-1073807239	VI_ERROR_INV_PROT	The protocol specified is invalid.
-1073807237	VI_ERROR_INV_SIZE	Invalid size of window specified.
-1073807232	VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.
-1073807231	VI_ERROR_NIMPL_OPER	The given operation is not implemented.

Instrument Driver Error Codes

Status	Description	Set By
0	No error: the call was successful	
-1200	Invalid syntax string	VISA Transition Library
-1201	Error finding instruments	VISA Transition Library
-1202	Unable to initialize interface or instrument	VISA Transition Library
-1205	Invalid Instrument Handle	VISA Transition Library
-1210	Parameter out of range	Instrument Driver VI
-1215	Error closing instrument	VISA Transition Library
-1218	Error getting attribute	VISA Transition Library
-1219	Error setting attribute	VISA Transition Library
-1223	Instrument identification query failed	Instrument Driver Initialize VI
-1224	Error clearing instrument	VISA Transition Library
-1225	Error triggering instrument	VISA Transition Library
-1226	Error polling instrument	VISA Transition Library
-1230	Error writing to instrument	VISA Transition Library

-1231	Error reading from instrument	VISA Transition Library
-1236	Error interpreting instrument response	Instrument Driver VI
-1240	Instrument timed out	VISA Transition Library
-1250	Error mapping VXI address	VISA Transition Library
-1251	Error unmapping VXI address	VISA Transition Library
-1252	Error accessing VXI address	VISA Transition Library
-1260	Function not supported by controller	VISA Transition Library
-1300	Instrument-specific error	
-13xx	Developer specified error codes	

GPIB Error Codes

Code	Name	Description
0	EDVR	Error connecting to driver.
1	ECIC	Command requires GPIB Controller to be CIC.
2	ENOL	Write detected no Listeners.
3	EADR	GPIB Controller not addressed correctly.
4	EARG	Invalid argument or arguments.
5	ESAC	Command requires GPIB Controller to be SC.
6	EABO	I/O operation aborted.
7	ENEB	Non-existent board.
8	EDMA	DMA Hardware error detected.
9	EBTO	DMA hardware mP bus timeout.
11	ECAP	No capability.
12	EFSO	File system operation error.
13	EOWN	Shareable board exclusively owned.
14	EBUS	GPIB bus error.
15	ESTB	Serial poll byte queue overflow.

16	ESRQ	SRQ stuck on.
17	ECMD	Unrecognized command.
19	EBNP	Board not present.
20	ETAB	Table error.
30	NADDR	No GPIB address input.
31	NSTRG	No string input (write).
32	NCNT	No count input (read).

Serial Port Error Codes

Code	Name	Description
61	EPAR	Serial port parity error.
62	EORN	Serial port overrun error.
63	EOFL	Serial port receive buffer overflow.
64	EFRM	Serial port framing error.
65	SPTMO	Serial port timeout, bytes not received at serial port.

Operation of the GPIB

This topic describes basic concepts you need to understand to operate the GPIB. It also contains a description of the physical and electrical characteristics of the GPIB and configuration requirements of the GPIB.

Types of Messages
Talkers, Listeners, and Controllers
Controller-In-Charge and System Controller
GPIB Signals and Lines
Physical and Electrical Characteristics

Types of Messages

The GPIB carries device-dependent messages and interface messages.

- Device-dependent messages, often called *data* or *data messages*, contain device-specific information such as programming instructions, measurement results, machine status, and data files.
- Interface messages manage the bus itself. They are usually called *commands* or *command messages*. Interface messages perform such tasks as initializing the bus, addressing and unaddressing devices, and setting device modes for remote or local programming.

Do not confuse the term *command* as used here with some device instructions, which can also be called commands. These device-specific instructions are actually data messages.

Talkers, Listeners, and Controllers

GPIB devices can be Talkers, Listeners, and/or Controllers. A digital voltmeter, for example, is a Talker and may be a Listener as well. A Talker sends data messages to one or more Listeners. The Controller manages the flow of information on the GPIB by sending commands to all devices.

The GPIB is like an ordinary computer bus, except that the computer has its circuit cards interconnected via a backplane bus, whereas the GPIB has stand-alone devices interconnected via a cable bus.

The role of the GPIB Controller is similar to the role of the CPU of a computer, but a better analogy is to the switching center of a city telephone system. The switching center (Controller) monitors the communications network (GPIB). When the center (Controller) notices that a party (device) wants to make a call (send a data message), it connects the caller (Talker) to the receiver (Listener).

The Controller addresses a Talker and a Listener before the Talker can send its message to the Listener. After the Talker transmits the message, the Controller may unaddress both devices.

Some bus configurations do not require a Controller. For example, one device may always be a Talker (called a Talk-only device) and there may be one or more Listen-only devices.

A Controller is necessary when you must change the active or addressed Talker or Listener. A computer usually handles the Controller function.

With the GPIB board and its software, your personal computer plays all three roles:

- Controllerto manage the GPIB
- Talkerto send data
- Listenerto receive data

Controller-In-Charge and System Controller

Although there can be multiple Controllers on the GPIB, only one Controller at a time is active or Controller-In-Charge (CIC). You can pass active control from the current CIC to an idle Controller. Only one device on the bus--the System Controller--can make itself the CIC. The GPIB board is usually the System Controller.

GPIB Signals and Lines

The interface system consists of 16 signal lines and 8 ground-return or shield-drain lines.

The 16 signal lines are divided into three groups:

- Eight data lines
- Three handshake lines
- Five interface management lines

Data Lines

The eight data lines, DIO1 through DIO8, carry both data and command messages. All commands and most data use the 7-bit ASCII or International Standards Organization (ISO) code set, in which case the eighth bit, DIO8, is unused or is used for parity.

Handshake Lines

Three lines asynchronously control the transfer of message bytes among devices. This process is called a three-wire interlocked handshake, and it guarantees that message bytes on the data lines are sent and received without transmission error.

NRFD (not ready for data)

NRFD indicates whether a device is ready to receive a message byte. All devices drive NRFD when they receive commands, and Listeners drive it when they receive data messages.

NDAC (not data accepted)

NDAC indicates whether a device has accepted a message byte. All devices drive NDAC when they receive commands, and Listeners drive it when they receive data messages.

DAV (data valid)

DAV tells whether the signals on the data lines are stable (valid) and whether devices can accept them safely. The Controller drives DAV when sending commands, and the Talker drives it when sending data messages.

Interface Management Lines

Five lines manage the flow of information across the interface.

ATN (attention)

The Controller drives ATN true when it uses the data lines to send commands and drives ATN false when a Talker can send data messages.

IFC (interface clear)

The System Controller drives the IFC line to initialize the bus and become CIC.

REN (remote enable)

The System Controller drives the REN line, which places devices in remote or local program mode.

SRQ (service request)

Any device can drive the SRQ line to asynchronously request service from the Controller.

EOI (end or identify)

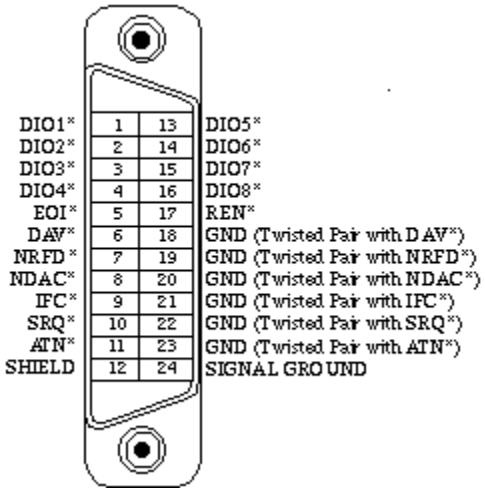
The EOI line has two purposes. The Talker uses the EOI line to mark the end of a message string. The Controller uses the EOI line to tell devices to respond in a parallel poll.

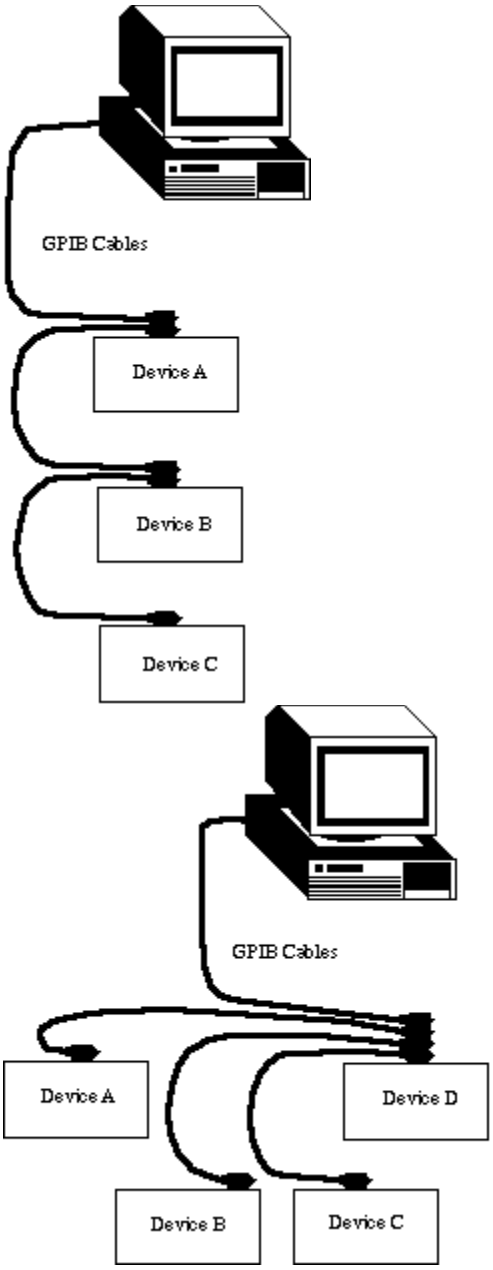
Physical and Electrical Characteristics

You usually connect devices with a cable assembly consisting of a shielded 24-conductor cable which has both a plug and a receptacle connector at each end. With this design, you can link devices in either a linear or a star configuration, or a combination of the two. See Figures B-1, B-2, and B-3.

The standard connector is the Amphenol or Cinch Series 57 *Microribbon* or *Amp Champ* type. You can use an adapter cable with a non-standard cable and/or connector for special interconnection applications.

The GPIB uses negative logic with standard transistor-transistor logic (TTL) level. When DAV is true, for example, it is a TTL low level (≤0.8 V), and when DAV is false, it is a TTL high level (≥ 2.0 V).





Configuration Requirements

To achieve the high data transfer rate for which the GPIB was designed, the physical distance between devices and the number of devices on the bus must be limited. The following restrictions are typical:

- A maximum separation of 4 m between any two devices and an average separation of 2 m over the entire bus.
- A maximum total cable length of 20 m.
- No more than 15 devices connected to each bus, with at least two-thirds powered on.

Contact National Instruments for bus extenders if your requirements exceed these limits.

Multiline Interface Messages

This topic lists multiline interface messages, which are commands that IEEE 488 defines. Multiline interface messages manage the GPIB--they perform tasks such as initializing the bus, addressing and unaddressing devices, and setting device modes for local or remote programming. These multiline interface messages are sent and received with ATN TRUE. This topic also includes the mnemonics and messages that correspond to the interface functions.

For more information on these messages, refer to the ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation.*

Multiline Interface Messages

Hex	Oct	Dec	ASCII	Msg
00	000	0	NUL	
01	001	1	SOH	GTL
02	002	2	STX	
03	003	3	ETX	
04	004	4	EOT	SDC
05	005	5	ENQ	PPC
06	006	6	ACK	
07	007	7	BEL	
08	010	8	BS	GET
09	011	9	HT	TCT
0A	012	10	LF	
0B	013	11	VT	
0C	014	12	FF	
0D	015	13	CR	
0E	016	14	SO	
0F	017	15	SI	
10	020	16	DLE	
11	021	17	DC1	LLO
12	022	18	DC2	

13	023	19	DC3	
14	024	20	DC4	DCL
15	025	21	NAK	PPU
16	026	22	SYN	
17	027	23	ETB	
18	030	24	CAN	SPE
19	031	25	EM	SPD
1A	032	26	SUB	
1B	033	27	ESC	
1C	034	28	FS	
1D	035	29	GS	
1E	036	30	RS	
1F	037	31	US	
20	040	32	SP	MLA0
21	041	33	!	MLA1
22	042	34	“	MLA2
23	043	35	#	MLA3
24	044	36	$	MLA4
25	045	37	%	MLA5
26	046	38	&	MLA6
27	047	39	’	MLA7
28	050	40	(	MLA8
29	051	41	)	MLA9
2A	052	42	*	MLA10
2B	053	43	+	MLA11
2C	054	44	,	MLA12
2D	055	45	-	MLA13
2E	056	46	.	MLA14

2F	057	47	/	MLA15
30	060	48	0	MLA16
31	061	49	1	MLA17
32	062	50	2	MLA18
33	063	51	3	MLA19
34	064	52	4	MLA20
35	065	53	5	MLA21
36	066	54	6	MLA22
37	067	55	7	MLA23
38	070	56	8	MLA24
39	071	57	9	MLA25
3A	072	58	:	MLA26
3B	073	59	;	MLA27
3C	074	60	<	MLA28
3D	075	61	=	MLA29
3E	076	62	>	MLA30
3F	077	63	?	UNL
40	100	64	@	MTA0
41	101	65	A	MTA1
42	102	66	B	MTA2
43	103	67	C	MTA3
44	104	68	D	MTA4
45	105	69	E	MTA5
46	106	70	F	MTA6
47	107	71	G	MTA7
48	110	72	H	MTA8
49	111	73	I	MTA9

4A	112	74	J	MTA10
4B	113	75	K	MTA11
4C	114	76	L	MTA12
4D	115	77	M	MTA13
4E	116	78	N	MTA14
4F	117	79	O	MTA15
50	120	80	P	MTA16
51	121	81	Q	MTA17
52	122	82	R	MTA18
53	123	83	S	MTA19
54	124	84	T	MTA20
55	125	85	U	MTA21
56	126	86	V	MTA22
57	127	87	W	MTA23
58	130	88	X	MTA24
59	131	89	Y	MTA25
5A	132	90	Z	MTA26
5B	133	91	[	MTA27
5C	134	92	\	MTA28
5D	135	93	]	MTA29
5E	136	94	^	MTA30
5F	137	95	_	UNT
60	140	96	`	MSA0,PPE
61	141	97	a	MSA1,PPE
62	142	98	b	MSA2,PPE
63	143	99	c	MSA3,PPE
64	144	100	d	MSA4,PPE

65	145	101	e	MSA5,PPE
66	146	102	f	MSA6,PPE
67	147	103	g	MSA7,PPE
68	150	104	h	MSA8,PPE
69	151	105	i	MSA9,PPE
6A	152	106	j	MSA10,PPE
6B	153	107	k	MSA11,PPE
6C	154	108	l	MSA12,PPE
6D	155	109	m	MSA13,PPE
6E	156	110	n	MSA14,PPE
6F	157	111	o	MSA15,PPE
70	160	112	p	MSA16,PPD
71	161	113	q	MSA17,PPD
72	162	114	r	MSA18,PPD
73	163	115	s	MSA19,PPD
74	164	116	t	MSA20,PPD
75	165	117	u	MSA21,PPD
76	166	118	v	MSA22,PPD
77	167	119	w	MSA23,PPD
78	170	120	x	MSA24,PPD
79	171	121	y	MSA25,PPD
7A	172	122	z	MSA26,PPD
7B	173	123	{	MSA27,PPD
7C	174	124	|	MSA28,PPD
7D	175	125	}	MSA29,PPD
7E	176	126	~	MSA30,PPD

7F	177	127	DEL

Message Definitions

Mnemonics	Definition
DCL	Device Clear
GET	Group Execute Trigger
GTL	Go To Local
LLO	Local Lockout
MLA	My Listen Address
MSA	My Secondary Address
MTA	My Talk Address
PPC	Parallel Poll Configure
PPD	Parallel Poll Disable
PPE	Parallel Poll Enable
PPU	Parallel Poll Unconfigure
SDC	Selected Device Clear
SPD	Serial Poll Disable
SPE	Serial Poll Enable
TCT	Take Control
UNL	Unlisten
UNT	Untalk

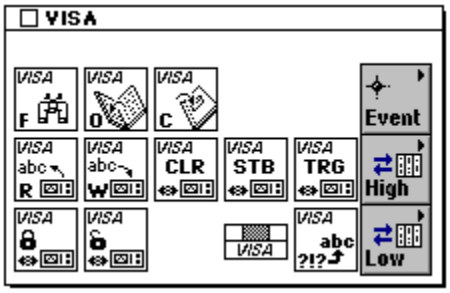
VISA Library Reference Function Descriptions

This topic contains descriptions of the VISA Library Reference operations and attributes.

Click here to access the VISA Overview topic.

The following figure shows the **VISA** palette, which you access by selecting **Functions**»**Instrument I/O**»**VISA**: The valid classes for the functions that appear on the main **VISA** palette are: Instr (default), GPIB Instr, Serial Instr, VXI/GPIB-VXI RBD Instr, and VXI/GPIB-VXI MBD Instr.

Click on one of the icons below for VI description information. You can also click on one of the text jumps below the icons to access VI descriptions.



VISA Clear
VISA Close
VISA Find Resource
VISA Lock
VISA Open
VISA Read
VISA Read STB
VISA Status Description
VISA Trigger
VISA Unlock
VISA Write

Subpalette Descriptions

Event Handling Function Descriptions
High Level Register Access Function Descriptions
Low Level Register Access Function Descriptions

VISA Clear

Performs an IEEE 488.1-style clear of the device. For VXI, this is the Word Serial Clear command; for GPIB systems, this is the Selected Device Clear command.

Note: **The Serial Instr class is not valid for VISA Clear.**

VISA session. See the VISA Session topic for more information.

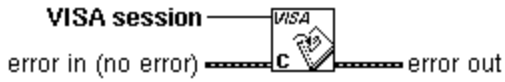
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Close

Closes a specified device session or event object. VISA Close accepts all available classes.



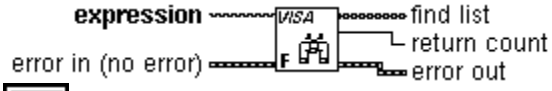
VISA session. See the VISA Session topic for more information.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Find Resource

Queries the system to locate the devices associated with a specified interface.



expression matches the value specified with the devices available for a particular interface. The description string specified sets the criteria to search an interface (GPIB, GPIB-VXI, VXI, All VXI, Serial or All) for existing devices. The description string format is shown in the following table:

Interface	Expression
GPIB	GPIB[0-9]*::? *INSTR
GPIB-VXI	GPIB-VXI?*INSTR
GPIB or GPIB-VXI	GPIB?*INSTR
VXI	VXI?*INSTR
All VXI	?*VXI[0-9]*::?*INSTR
Serial	ASRL[0-9]*::?*INSTR
All	?*INSTR

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

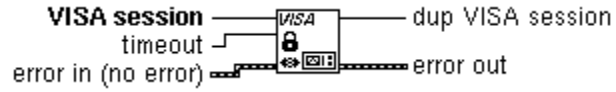
find list. An array of strings, each string specifying one interface found by the function.

return count. Specifies the number of matches found; (the number of strings in the find list array).

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Lock

Establishes exclusive access to the specified source.



VISA session. See the VISA Session topic for more information.

timeout. Period of time to wait for access to lock.

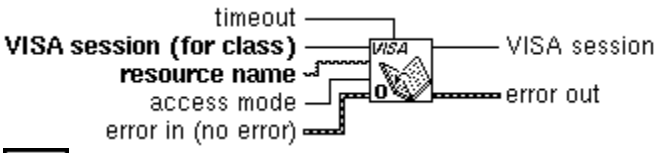
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Open

Opens a session to the specified device and returns a session identifier that can be used to call any other operations of that device.



timeout should be set to zero (0).

VISA session is wired to indicate the class of the session being opened. It need not be a valid session, only the class of the session is important. See the VISA Session topic for more information.

resource name. Unique symbolic name of a resource.

access mode should be set to zero (0).

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

VISA session is the session opened by the function. It must match the class of the input **VISA session**. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

The following table shows the grammar for the address string. Optional parameters are shown in square brackets ([ ]).

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

The GPIB keyword can be used to establish communication with a GPIB device. The GPIB-VXI keyword is used for a GPIB-VXI controller. The VXI keyword is used for VXI instruments via either embedded or MXIbus controllers. The Serial keyword is used to establish communication with an asynchronous serial (such as RS-232) device.

The INSTR keyword specifies a VISA resource of the type INSTR.

The following table shows the default value for optional parameters:

Optional Parameter	Default Value
board	0
secondary address	none

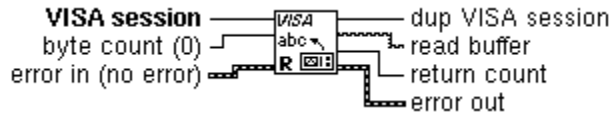
The following table shows examples of address strings:

Address String	Description
GPIB::1::0::INSTR	A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0.
GPIB-VXI::9::INSTR	A VXI device at logical address 9 in a GPIB-VXI controlled VXI system.
VXI0::1::INSTR	A VXI device at logical address 1 in VXI interface VXI0.
ASRL0::INSTR	A serial device located on port 0.

See the VISA Close description for more information.

VISA Read

Reads data from a device. On UNIX platforms data is read synchronously; on all other platforms data is read asynchronously.



VISA session. See the VISA Session topic for more information.

byte count (0). Number of bytes to be read.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

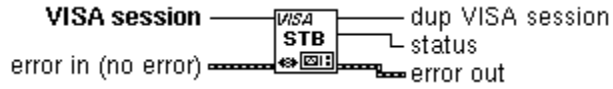
read buffer contains the data read from the device.

return count contains the number of bytes actually read.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Read STB

Reads a service request status from a message-based device. For example, on the IEEE 488.2 interface, the message is read by polling devices. For other types of interfaces, a message is sent in response to a service request to retrieve status information. If the status information is only one byte long, the most significant byte is returned with the zero value.



Note: **The Serial Instr class is not valid for VISA Read STB.**

VISA session. See the VISA Session topic for more information.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

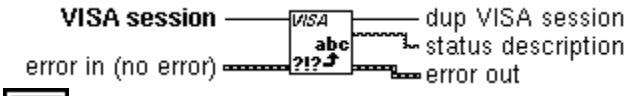
dup VISA session. See the VISA Session topic for more information.

status contains the status byte read from the instrument.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Status Description

Retrieves a user-readable string that describes the status code presented in **error in**.



VISA session. See the VISA Session topic for more information.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

status description contains the user-readable string interpretation of the status code passed to the operation.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Trigger

Asserts a software or hardware trigger, depending on the interface type.

Note: **The Serial Instr class is not valid for VISA Assert Trigger.**

VISA session. See the VISA Session topic for more information.

protocol is the trigger protocol to use during assertion. Valid protocol values are:

0: default
1: on
2: off
5: sync

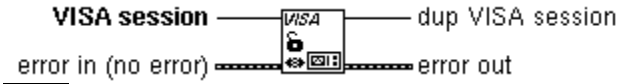
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Unlock

Relinquishes the lock previously obtained using the VISA Lock function.



VISA session. See the VISA Session topic for more information.

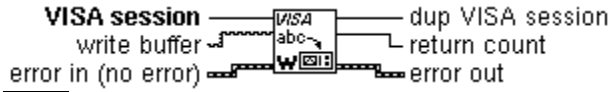
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Write

Writes data to the device. On UNIX platforms data is written synchronously; on all other platforms data is written asynchronously.



VISA session. See the VISA Session topic for more information.

write buffer contains the data to be written to the device.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

return count contains the actual number of bytes written.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

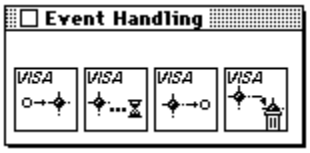
Event Handling Function Descriptions

Click here to access the VISA Overview topic.

The following topics describe the VISA Event Handling functions. Valid classes for these functions are: Instr (default), GPIB Instr, Serial Instr, VXI/GPIB-VXI RBD Instr, and VXI/GPIB-VXI MBD Instr.

To access the VISA Event Handling functions, select **Functions»VISA»Event Handling**:

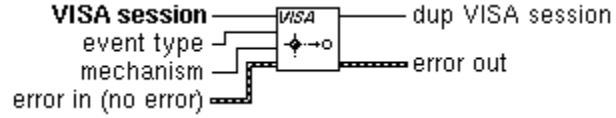
Click on one of the icons below for VI description information. You can also click on one of the text jumps below the icons to access VI descriptions.



VISA Disable Event
VISA Discard Events
VISA Enable Event
VISA Wait On Event

VISA Disable Event

Disables servicing of an event. This operation prevents new event occurrences from being queued. However, event occurrences already queued are not lost; use VISA Discard Events if you want to discard queued events.



VISA session. See the VISA Session topic for more information.

event type is the logical event identifier. Special values are:

Trigger:	0xBFFF200A
VXI Signal:	0x3FFF2020
Service Request:	0x3FFF200B
All Events:	0x3FFF7FFF

mechanism. Specifies the event handling mechanism to be disabled. Currently only accepts `VI_QUEUE` (1).

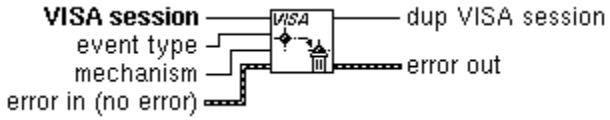
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Discard Events

Discards all pending occurrences of the specified event types and mechanisms from the specified session.



VISA session. See the VISA Session topic for more information.

event type is the logical event identifier. Special values are:

Trigger:	0xBFFF200A
VXI Signal:	0x3FFF2020
Service Request:	0x3FFF200B
All Events:	0x3FFF7FFF

mechanism. Specifies the event handling mechanism to be disabled. Currently only accepts VI_QUEUE (1).

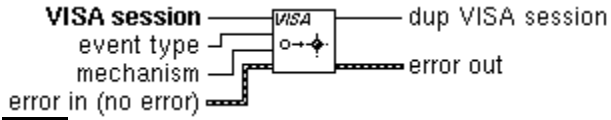
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Enable Event

Enables notification of a specified event.



VISA session. See the VISA Session topic for more information.

event type is the logical event identifier. Special values are:

Trigger:	0xBFFF200A
VXI Signal:	0x3FFF2020
Service Request:	0x3FFF200B

mechanism. Specifies the event handling mechanism to be disabled. Currently only accepts VI_QUEUE (1).

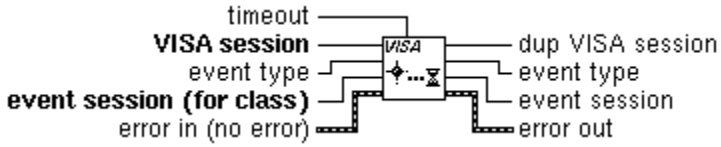
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Wait On Event

Suspends execution of a thread of application and waits for an event Event Type for a time period not to exceed that specified by timeout. Refer to individual event descriptions for context definitions. If the specified event type is All Events, the operation waits for any event that is enabled for the given session.



 **VISA session**. See the VISA Session topic for more information.

 **event session (for class)** specifies the class of event being waited for.

 **timeout**. Period of time to wait for the event.

 **event type** is the logical event identifier. Special values are:

Trigger:	`0xBFFF200A`
VXI Signal:	`0x3FFF2020`
Service Request:	`0x3FFF200B`

 **error in (no error**). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

 **dup VISA session**. See the VISA Session topic for more information.

 **event type** identifies the event type received if the wait was successful.

 **event session** is valid if the wait was successful. Can be wired to an attribute node to get further information about the event and should be wired to the VISA Close function when examination of the event is complete.

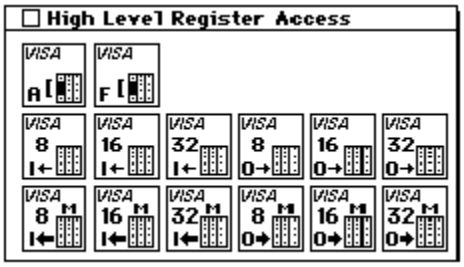
 **error out**. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

High Level Register Access Function Descriptions

Click here to access the VISA Overview topic.

The following topics describe the VISA High Level Register Access functions. Valid classes for these functions are: Instr (default), VXI/GPIB-VXI RBD Instr, and VXI/GPIB-VXI MBD Instr. You can access these functions by selecting **Functions**»**VISA**»**High Level Register Access**:

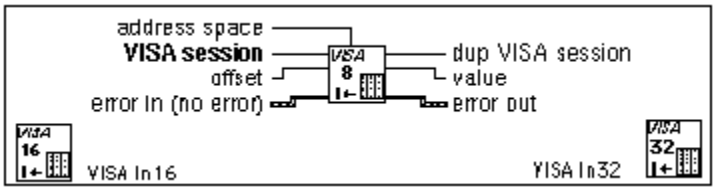
Click on one of the icons below for VI description information. You can also click on one of the text jumps below the icons to access VI descriptions.



VISA Move In8 / Move In16 / Move In32
VISA Memory Allocation
VISA Memory Free
VISA Move In8 / Move In16 / Move In32
VISA Move Out8 / Move Out16 / Move Out32
VISA Out8 / Out16 / Out32

VISA In8 / In16 / In32

Reads in 8-bits, 16-bits, or 32-bits of data, respectively, from the specified memory space (assigned memory base + offset).



VISA session. See the VISA Session topic for more information.

address space. Specifies the address space without requiring VISA Map Address to be called. The following table lists the valid entries for specifying address space:

Value	Description
A16 SPACE (1)	Addresses the A16 address space of VXI/MXI bus.
A24 SPACE (2)	Addresses the A24 address space of VXI/MXI bus.
A32 SPACE (3)	Addresses the A32 address space of VXI/MXI bus.

offset. Offset (in bytes) of the device to read from. It is the offset address relative to the devices allocated address base for the corresponding specified address space. For example, if **address space**

specifies A16 SPACE, then **offset** specifies the offset from the logical address base address of the VXI device specified. If **address space** specifies A24 or A32 SPACE, **offset** specifies the offset from the base address of the VXI devices memory space allocated by the VXI Resource Manager within VXI A24 or A32 SPACE.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

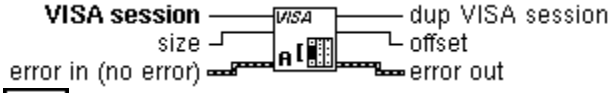
dup VISA session. See the VISA Session topic for more information.

value contains the data read from bus (8-bits for In8, 16-bits for In16, 32-bits for In32; 8-bits is shown).

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Memory Allocation

Returns an offset into a devices region that has been allocated for use by the session. The memory can be allocated on either the device itself or on the computers system memory.



VISA session. See the VISA Session topic for more information.

size specifies the size of the allocation.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

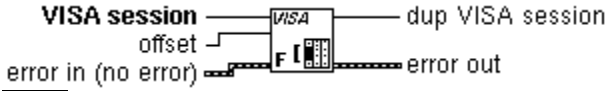
dup VISA session. See the VISA Session topic for more information.

offset returns the offset which specifies where the memory was allocated.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Memory Free

Frees the memory previously allocated by the VISA Memory Allocation function.



VISA session. See the VISA Session topic for more information.

offset specifies the memory previously allocated by VISA Memory Allocation.

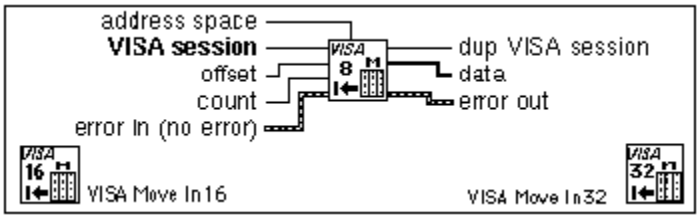
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Move In8 / Move In16 / Move In32

Moves a block of data from device memory to local memory in accesses of 8-bits, 16-bits, or 32-bits, respectively.



address space. Specifies the address space to map. For more information, see the table listed under **address space**, in the VISA In function.

VISA session. See the VISA Session topic for more information.

offset. Offset (in bytes) of the device to read from. It is the offset address relative to the devices allocated address base for the corresponding specified address space. For example, if **address space** specifies A16 SPACE, then **offset** specifies the offset from the logical address base address of the VXI device specified. If **address space** specifies A24 or A32 SPACE, **offset** specifies the offset from the base address of the VXI devices memory space allocated by the VXI Resource Manager within VXI A24 or A32 SPACE.

count. Counts the number of data items to move.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

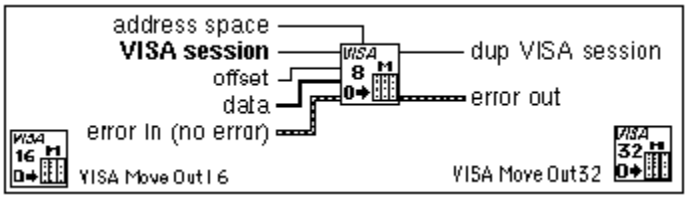
dup VISA session. See the VISA Session topic for more information.

data is the data which is being moved (8-bits for Move In8, 16-bits for Move In16, 32-bits for Move In32; 8-bits is shown).

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Move Out8 / Move Out16 / Move Out32

Moves a block of data from local memory to device memory in accesses of 8-bits, 16-bits, or 32-bits, respectively.



address space. Specifies the address space to map. For more information, see the table listed under the VISA In8 / In16 / In32 function.

VISA session. See the VISA Session topic for more information.

offset. Offset (in bytes) of the device to write to. It is the offset address relative to the devices allocated address base for the corresponding specified address space. For example, if **address space** specifies A16 SPACE, then **offset** specifies the offset from the logical address base address of the VXI device specified. If **address space** specifies A24 or A32 SPACE, **offset** specifies the offset from the base address of the VXI devices memory space allocated by the VXI Resource Manager within VXI A24 or A32 SPACE.

data is the data which is being moved (8-bits for Move Out8, 16-bits for Move Out16, 32-bits for Move Out32; 8-bits is shown).

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

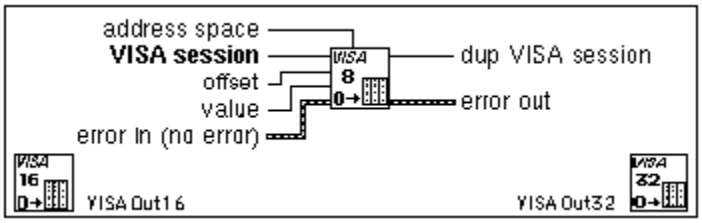
dup VISA session. See the VISA Session topic for more information.

count. Counts the number of data items to move.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Out8 / Out16 / Out32

Writes 8-bits, 16-bits, or 32-bits of data, respectively, to the specified memory space (assigned memory base + offset).



VISA session. See the VISA Session topic for more information.

address space. Specifies the address space to map. For more information, see the table listed under **address space**, under the VISA In function.

 **offset**. Offset (in bytes) of the device to write to. It is the offset address relative to the devices allocated address base for the corresponding specified address space. For example, if **address space** specifies A16 SPACE, then **offset** specifies the offset from the logical address base address of the VXI device specified. If **address space** specifies A24 or A32 SPACE, **offset** specifies the offset from the base address of the VXI devices memory space allocated by the VXI Resource Manager within VXI A24 or A32 SPACE.

 . **value** contains the data to write to bus (8-bits for Out8, 16-bits for Out16, 32-bits for Out32; 8-bits is shown).

 **error in (no error**). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

 **dup VISA session**. See the VISA Session topic for more information.

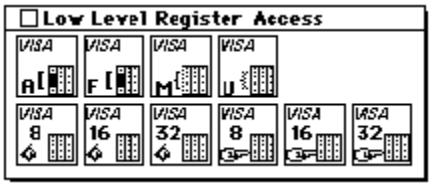
 **error out**. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

Low Level Register Access Function Descriptions

Click here to access the VISA Overview topic.

The following topic describes the VISA Low Level Register Access functions. Valid classes for these functions are: Instr (default), VXI/GPIB-VXI RBD Instr, and VXI/GPIB-VXI MBD Instr.You can access these functions by selecting **Functions**»**VISA**»**Low Level Register Access** functions:

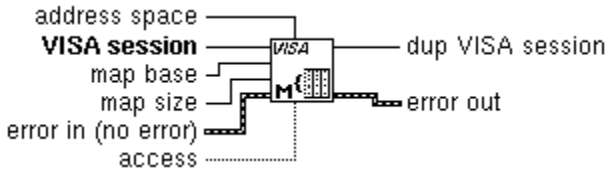
Click on one of the icons below for VI description information. You can also click on one of the text jumps below the icons to access VI descriptions.



VISA Map Address
VISA Memory Allocation
VISA Memory Free
VISA Peek8 / Peek16 / Peek32
VISA Poke8 / Poke16 / Poke32

VISA Map Address

Maps in a specified memory space.



VISA session. See the VISA Session topic for more information.

address space. Specifies the **address space** to map. For more information, see the table listed under **address space**, in the VISA In function.

map base contains the offset (in bytes) of the memory to be mapped.

map size contains the amount of memory to map.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

access should be set to false (F).

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Memory Allocation

Returns an offset into a devices region that has been allocated for use by the session. The memory can be allocated on either the device itself or on the computers system memory.

VISA session. See the VISA Session topic for more information.

size specifies the size of the allocation.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

offset returns the offset which specifies where the memory was allocated.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Memory Free

Frees the memory previously allocated by the VISA Memory Allocation function.

VISA session. See the VISA Session topic for more information.

offset specifies the memory previously allocated by VISA Memory Allocation.

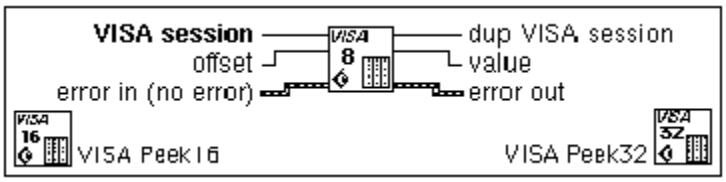
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Peek8 / Peek16 / Peek32

Reads an 8-bit, 16-bit, or 32-bit value, respectively, from the specified address.



VISA session. See the VISA Session topic for more information.

offset contains the offset (in bytes) from the register location originally mapped in the mapped memory space.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

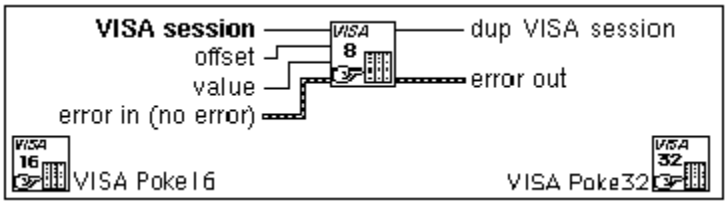
dup VISA session. See the VISA Session topic for more information.

value contains the read data (8-bits for VISA Peek8, 16-bits for Peek16, 32-bits for Peek32; 8-bits is shown).

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Poke8 / Poke16 / Poke32

Writes an 8-bit, 16-bit, or 32-bit value, respectively, to the specified address.



VISA session. See the VISA Session topic for more information.

offset contains the offset (in bytes) from the register location originally mapped in the mapped memory space.

value contains the value to write (8-bits for Poke8, 16-bits for Poke16, 32-bits for Poke32; 8-bits is shown).

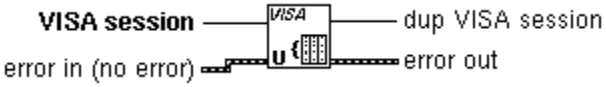
error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Unmap Address

Unmaps memory space previously mapped by VISA Map Address.



VISA session. See the VISA Session topic for more information.

error in (no error). See the Error Reporting topic for a detailed description of **error in**. See VISA Error In/Error Out for error information specific to VISA.

dup VISA session. See the VISA Session topic for more information.

error out. See the Error Reporting topic for a detailed description of **error out**. See VISA Error In/Error Out for error information specific to VISA.

VISA Attribute Descriptions

Click here to access the VISA Overview topic.

Each attribute description includes the attributes range of values, default value, and access privilege. *Local* applies the current session only. *Global* refers to all sessions to the same VISA resource.

Fast Data Channel Mode
Fast Data Channel Number
Fast Data Channel Pairs
Fast Data Channel Signal Enable
GPIB Primary Address
GPIB Secondary Address
I/O Protocol
Immediate Servant
Increment Destination Count
Increment Source Count
Interface Number
InterfaceType
Mainframe Logical Address
Manufacturer ID
Maximum Queue Length
Model Code
Resource Lock State
Resource Manufacturer Identification
Resource Manufacturer Name
Resource Name
Send End Enable
Slot
Suppress End Enable
Termination Character
Termination Character Enable
Timeout Value
Trigger Identifier
User Data
Version of Implementation
Version of Specification
VXI Commander Logical Address
VXI Logical Address
VXI Memory Address Space
VXI Memory Base Address
VXI Memory Size
Window Access
Window Base Address
Window Size

Fast Data Channel Mode

Specifies which FDC mode to use (either normal or stream mode).
Range: FDC Normal (1)/FDC Stream (2)
Default: FDC Normal (1)
Access Privilege: Read/Write Local

Fast Data Channel Number

Determines which fast data channel (FDC) will be used to transfer the buffer.

Range: 0 to 7
Default: N/A
Access Privilege: Read/Write Local

Fast Data Channel Pairs

Specifies use of a channel pair for transferring data; (otherwise, only one channel will be used).

Range: True/False
Default: False
Access Privilege: Read/Write

Fast Data Channel Signal Enable

Lets the servant send a signal when control of the FDC channel is passed back to the commander. This action frees the commander from having to poll the FDC header while engaging in an FDC transfer.

Range: True/False
Default: False
Access Privilege: Read/Write Local

GPIB Primary Address

Specifies the primary address of the GPIB device used by the given session.

Range: 0 to 30
Default: N/A
Access Privilege: Read Only Global

GPIB Secondary Address

Specifies the secondary address of the GPIB device used by the given session.

Range: 0 to 31; FFFF for no secondary address
Default: N/A
Access Privilege: Read Only Global

I/O Protocol

Specifies which protocol to use. In VXI systems you can choose between normal word serial or fast data channel (FDC). In GPIB, you can choose between normal and high-speed (HS488) data transfers.

Range: GPIB: Normal (1)/HS488 (3)
VXI, GPIB-VXI: Normal (1)/FDC(2)
Default: Normal (1)
Access Privilege: Read/Write Local

Immediate Servant

Determines if the VXI device is an immediate servant of the local controller.

Range: True/False
Default: N/A

Access Privilege: Read Only Global

Increment Destination Count

 Specifies the number of elements by which to increment the destination address on block move operations.

Range: 0, 1
Default: 1
Access Privilege: Read/Write Local

Increment Source Count

 Specifies the number of elements by which to increment the source address on block move operations.

Range: 0,1
Default: 1
Access Privilege: Read/Write

Interface Number

 Specifies the board number for the given interface.

Range: 0h to FFFFh
Default: 0
Access Privilege: Read Only Global

InterfaceType

 Specifies the interface type of the given session.

Range: GPIB (1), GPIB-VXI (3), VXI (2), Serial (4)
Default: N/A
Access Privilege: Read Only Global

Mainframe Logical Address

 Specifies the lowest logical address in the mainframe. If the logical address is not known, UNKNOWN LA is returned.

Range: 0 to 255; UNKNOWN LA (-1)
Default: N/A
Access Privilege: Read Only Global

Manufacturer ID

 The manufacturer identification number of the VXIbus device.

Range: 0h to FFFh
Default: N/A
Access Privilege: Read Only Global

Maximum Queue Length

 Specifies the maximum number of events that can be queued at any time on the given session. This attribute is Read/Write until the first time Enable Event is called on a session. Thereafter, this attribute is Read Only.

Range: 1h to FFFFFFFFh
Default: 50
Access Privilege: Read/Write Local

Model Code

Specifies the model code for the VXIbus device.

Range: 0h to FFFFh
Default: N/A
Access Privilege: Read Only Global

Resource Lock State

{bmc UINT32i.BMP} Reflects the current locking state of the resource that is associated with the given session.

Range: No Lock (0), Exclusive Lock (1), Shared Lock (2)
Default: No Lock (0)
Access Privilege: Read Only Global

Resource Manufacturer Identification

A value corresponding to the VXI manufacturer ID of the manufacturer that created the implementation.

Range: 0h to 3FFFh
Default: N/A
Access Privilege: Read Only Global

Resource Manufacturer Name

{bmc ABCi.BMP} A string that corresponds to the VXI manufacturer name of the manufacturer that created the implementation.

Range: N/A
Default: N/A
Access Privilege: Read Only Global

Resource Name

{bmc ABCi.BMP} Unique identifier for a resource.

Range: N/A
Default: N/A
Access Privilege: Read Only Global

The address structure is shown in the following table. Optional parameters are shown in square brackets:

Interface	Grammar
GPIB	GPIB[board]::primary address[::secondary address] [::INSTR]
GPIB-VXI	GPIB-VXI[board]::VXI logical address[::INSTR]

VXI	VXI[board]::VXI logical address[::INSTR]
Serial	ASRL[board][::INSTR]

Address String	Description
GPIB::1::0::INSTR	A GPIB device at primary address 1 and secondary address 0 in GPIB interface 0.
GPIB-VXI::9::INSTR	A VXI device at logical address 9 in a GPIB-VXI controlled VXI system.
VXI0::1::INSTR	A VXI device at logical address 1 in VXI interface VXI0.
ASRL0::INSTR	A serial device located on port 0.

Send End Enable

Specifies whether to assert END during the transfer of the last byte of the buffer.

Range: True/False
Default: True
Access Privilege: Read/Write Local

Slot

Specifies the physical slot location of the VXIbus device. If the slot number is not known, UNKNOWN SLOT is returned.

Range: 0-12; UNKNOWN SLOT (-1)
Default: N/A
Access Privilege: Read Only Global

Suppress End Enable

Specifies whether to suppress the END bit termination. If this attribute is set to TRUE, the END bit does not terminate read operations. If this attribute is set to FALSE, the END bit terminates read operations.

Range: True/False
Default: False
Access Privilege: Read/Write Local

Termination Character

{bmc UINT8.BMP} The termination character. When the termination character is read and Termination Character Enable is enabled during a read operation, the read operation terminates. See the description for Termination Character Enable listed below.

Range: 0 to FFh
Default: 0Ah (line feed)
Access Privilege: Read/Write Local

Termination Character Enable

 Determines whether the read operation should terminate when a termination character is received.

Range: True/False
Default: False
Access Privilege: Read/Write

Timeout Value

 Specifies the timeout value to use (in milliseconds) when accessing the device associated with the given session. A timeout value of TMO IMMEDIATE means that operations should never wait for the device to respond. A timeout value of TMO INFINITE disables the timeout mechanism.

Range: IMMEDIATE (0); INFINITE (FFFFFFFFh)
Entire range 1 to FFFFFFFEh
Default: 2000
Access Privilege: Read/Write Local

Trigger Identifier

 Identifier for the current triggering mechanism.

Range: GPIB: VI_TRIG_SW (-1)
GPIB-VXI, VXI: VI_TRIG_SW (-1);
VI_TRIG_TTL0 to VI_TRIG_TTL7 (0-7);
VI_TRIG_ECL0 to VI_TRIG_ECL1 (8-9)
Default: VI_TRIG_SW (-1)
Access Privilege: Read/Write Local

Note: **Trigger ID is Read/Write when the corresponding session is not enabled to receive trigger events. When the session is enabled to receive trigger events, the attribute is Read Only.**

User Data

 Used privately by the application for a particular session. This data is not used by VISA for any purposes. It is provided to the application for its own use.

Range: 0h to FFFFFFFFh
Default: N/A
Access Privilege: Read/Write

Version of Implementation

 Uniquely identifies each of the different revisions or implementations of a resource. This attribute value is defined by the individual manufacturer and increments with each new revision. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

Range: 0h to FFFFFFFFh
Default: N/A
Access Privilege: Read Only Global

Version of Specification

 Uniquely identifies the version of the VISA specification to which the implementation is compliant. The format of the value has the upper 12 bits as the major number of the version, the next lower 12 bits

as the minor number of the version, and the lowest 8 bits as the sub-minor number of the version.

Range: 0h to FFFFFFFFh
Default: 00100000h
Access Privilege: Read Only Global

VXI Commander Logical Address

The logical address of the commander of the VXI device.

Range: 0 to 255
Default: N/A
Access Privilege: Read Only Global

VXI Logical Address

Specifies the logical address of the VXI device used by the given session.

Range: 0 to 255
Default: N/A
Access Privilege: Read Only Global

VXI Memory Address Space

Specifies the VXIbus address space used by the device. The three types are A16 only, A16/A24, or A16/A32 memory address space.

Range: A16 (1), A24 (2), A32 (3)
Default: A16 (1)
Access Privilege: Read Only Global

VXI Memory Base Address

Specifies the base address of the device in VXIbus memory address space. This base address is applicable to A24 or A32 address space.

Range: 0h to FFFFFFFFh
Default: N/A
Access Privilege: Read Only Global

VXI Memory Size

Specifies the size of memory requested by the device in VXIbus address space.

Range: 0h to FFFFFFFFh
Default: N/A
Access Privilege: Read Only

Window Access

Specifies the modes in which the current window may be accessed.

Range: NMAPPED (1), USE_OPERS (2)
Default: NMAPPED (1)
Access Privilege: Read Only Local

Window Base Address

Specifies the base address of the interface bus to which this window is mapped.

Range: 0h to FFFFFFFFh
Default: N/A
Access Privilege: Read Only Local

Window Size

Specifies the size of the region mapped to this window.
Range: 0h to FFFFFFFFh
Default: N/A
Access Privilege: Read Only

VISA Overview

Click here to access the VISA Library Reference Function Descriptions.

VISA Functions
VISA Attribute Node

VISA Functions

VISA (Virtual Instrument Software Architecture) is a single interface library for controlling VXI, GPIB, RS-232, and other types of instruments. The VISA Library provides a standard set of I/O routines used by all LabVIEW instrument drivers. Using the VISA functions, you can construct a single instrument driver VI which controls a particular instrument model across different I/O interfaces.

An instrument descriptor is passed to the VISA Open function in order to select which kind of I/O will be used to communicate with the instrument. Once the session with the instrument is open, functions such as VISA Read and VISA Write perform the instrument I/O activities in a generic manner such that the program is not tied to any specific GPIB or VXI functions. Such an instrument driver is considered to be interface independent and can be used as is in different systems.

Instrument drivers which use the VISA functions perform activities specific to the instrument, not to the communication interface. This creates more opportunities for using the instrument driver in many diverse situations.

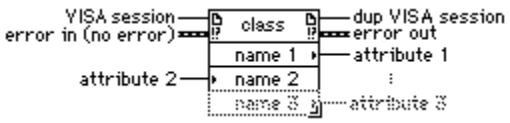
The following table lists the VISA functions:

VISA Function	Description
VISA Trigger	Asserts a software or hardware trigger.
VISA Clear	Performs an IEEE 488.1-style clear of the device.
VISA Close	Closes a specified device session.
VISA Disable Event	Disables servicing of an event.
VISA Discard Events	Discards all pending occurrences of the specified event types and mechanisms from the specified session.
VISA Enable Event	Enables notification of a specified event.
VISA Find Resource	Queries the system to locate the devices associated with a specified interface.
VISA In 8/16/32	Reads in 8-bits, 16-bits, or 32-bits of data, respectively, from the specified memory space.
VISA Lock	Establishes exclusive access to the specified resource.
VISA Map Address	Maps in a specified memory space.
VISA Memory Allocation	Returns an offset into a devices region that has been allocated for use by the session.
VISA Memory Free	Frees the memory previously allocated by the VISA

	Memory Allocation function.
VISA Move In 8/16/32	Moves a block of data from device memory to local memory in accesses of 8-bits, 16-bits, or 32-bits, respectively.
VISA Move Out 8/16/32	Moves a block of data from local memory to device memory in accesses of 8-bits, 16-bits, or 32-bits, respectively.
VISA Open	Opens a session to the specified device and returns a session identifier.
VISA Out 8/16/32	Writes 8-bits, 16-bits, or 32-bits of data, respectively, to the specified memory space.
VISA Peek 8/16/32	Reads an 8-bit, 16-bit, or 32-bit value, respectively, from the mapped memory region.
VISA Poke 8/16/32	Writes an 8-bit, 16-bit, or 32-bit value, respectively, to the mapped memory region.
VISA Read	Reads data from a device.
VISA Read STB	Reads a service request status from a message-based device.
VISA Status Description	Retrieves a user-readable string that describes the status code.
VISA Unlock	Relinquishes the lock previously obtained using the VISA Lock function.
VISA Unmap Address	Unmaps memory space previously mapped by VISA Map Address.
VISA Wait On Event	Suspends execution of a thread of application and waits for an event of the specified type.
VISA Write	Writes data to the device.

VISA Attribute Node

This topic describes the VISA Library attributes. The VISA Attribute Node gets and/or sets the indicated attributes. The node is resizable; evaluation starts from the top and proceeds downward until an error, or until the final evaluation, occurs.



To access the attribute node, select **Functions**»**Instrument I/O**»**VISA**. Then select the attribute node icon located on the bottom row of the **VISA** palette.

The VISA Attribute Node only displays attributes for the class of the session that is wired to it. You can change the class of a VISA Attribute Node as long as you have not wired it to a **VISA session**. Once a **VISA session** is wired to a VISA Attribute Node, it adapts to the class of the session and any displayed attributes which are not valid for that class become invalid (this is indicated by turning the attribute item black).

The following table lists the VISA attributes:

VISA Attribute	Description
Fast Data Channel Mode	Specifies which fast data channel (FDC) transfer mode to use.
Fast Data Channel Number	Determines which FDC will be used to transfer the buffer.
FDC Pairs	Specifies use of a channel pair for transferring data.
FDC Signal Enable	Lets the servant send a signal when control of the FDC channel is passed back to the commander.
GPIB Primary Address	Specifies the primary address of the GPIB device used by the given session.
GPIB Secondary Address	Specifies the secondary address of the GPIB device used by the given session.
IO Protocol	Specifies which protocol to use.
Immediate Servant	Determines if the VXI device is an immediate servant of the local controller.
Increment Destination Count	Specifies the number of elements by which to increment the destination address on block move operations.
Increment Source Count	Specifies the number of elements by which to increment the source address on block move operations.
Interface Number	Specifies the board number for the given interface.
Interface Type	Specifies the interface type of the given session.
Mainframe Logical Address	Specifies the lowest logical address in the mainframe.
Manufacturer ID	The manufacturer identification number of the VXIbus device.
Maximum Queue Length	Specifies the maximum number of events that can be queued at any time on the given session.
Model Code	Specifies the model code for the VXIbus device.
Resource Lock State	Reflects the current locking state of the resource that is associated with the given session.
Resource Manufacturer ID	A value corresponding to the VXI manufacturer ID of the

	manufacturer that created the implementation.
Resource Manufacturer Name	A string that corresponds to the VXI manufacturer name of the manufacturer that created the implementation.
Resource Name	Unique identifier for a resource.
Send End Enable	Specifies whether to assert END during the transfer of the last byte of the buffer.
Slot	Specifies the physical slot location of the VXIbus device.
Suppress End Enable	Specifies whether to suppress the END bit termination.
Termination Character	The termination character.
Termination Character Enable	Determines whether the read operation should terminate when a termination character is received.
Timeout Value	Specifies the timeout value to use when accessing the device associated with the given session.
Trigger Identifier	Identifier for the current triggering mechanism.
User Data	Used privately by the application for a particular session.
Version of Implementation	Uniquely identifies the current revision or implementation of a resource.
Version of Specification	Uniquely identifies the version of the VISA specification to which the implementation is compliant.
VXI Commander LA	The logical address of the commander of the VXI device.
VXI Logical Address	Specifies the logical address of the VXI device used by the given session.
VXI Memory Address Space	Specifies the VXIbus address space used by the device.
VXI Memory Base Address	Specifies the base address of the device in VXIbus memory address space.
VXI Memory Size	Specifies the size of memory requested by the device in VXIbus address space.
Window Access	Specifies the modes in which the current window may be accessed.
Window Base Address	Specifies the base address of the interface bus to which this window is mapped.
Window Size	Specifies the size of the region mapped to this window.

VISA Session

VISA session is a unique logical identifier to a session. It is produced by the VISA Open function and used by the VISA primitives. **dup VISA session** is the **VISA session** passed to a function. The dup simplifies dataflow programming and is similar to the dup file refnums provided by file I/O functions.

The **VISA session** drops by default with class *Instr*. You can change the class by popping up on it at edit time. The following classes are currently supported:

- Instr
- GPIB Instr
- VXI/GPIB-VXI RBD Instr
- VXI/GPIB-VXI MBD Instr
- Serial Instr
- Generic Event
- Service Request Event
- Trigger Event
- VXI Signal Event

Note: **The Generic Event, Service Request Event, Trigger Event, and VXI Signal Event classes work only with the VISA Close function and the VISA Attribute Node.**

VISA functions vary in the class of **VISA session** which can be wired to them. The valid classes for each function are indicated in the documentation. For example, the functions on the **High Level-** and **Low Level Register Access** palettes do not accept VISA sessions of class GPIB Instr or Serial Instr. If you wire a **VISA session** to a function that does not accept the class of the session, or if you wire two VISA sessions of differing classes together, your diagram will be broken and the error will be reported as a *Class Conflict*.

VISA Event Handling Function Subpalette

Event Handling Function Descriptions

Low Level Register Access Subpalette

Low Level Register Access Function Descriptions

High Level Register Access Subpalette

High Level Register Access Function Descriptions

VISA Map Address Function

VISA Map Address

VISA Unmap Address Function

VISA Unmap Address

VISA Peek 8 Function

VISA Peek8 / Peek16 / Peek32

VISA Peek 16 Function

VISA Peek8 / Peek16 / Peek32

VISA Peek 32 Function

VISA Peek8 / Peek16 / Peek32

VISA Poke 8 Function

VISA Poke8 / Poke16 / Poke32

VISA Poke 16 Function

VISA Poke8 / Poke16 / Poke32

VISA Poke 32 Function

VISA Poke8 / Poke16 / Poke32

VISA In 8 Function

VISA In8 / In16 / In32

VISA In 16 Function

VISA In8 / In16 / In32

VISA In 32 Function

VISA In8 / In16 / In32

VISA Out 8 Function

VISA Out8 / Out16 / Out32

VISA Out 16 Function

VISA Out8 / Out16 / Out32

VISA Out 32 Function

VISA Out8 / Out16 / Out32

VISA Enable Event Function

VISA Enable Event

VISA Discard Events Function

VISA Discard Events

VISA Disable Event Function

VISA Disable Event

VISA Wait on Event Function

VISA Wait On Event

VISA Memory Allocation Function

VISA Memory Allocation

VISA Memory Free Function

VISA Memory Free

VISA Memory Allocation Function

VISA Memory Allocation

VISA Memory Free Function

VISA Memory Free

VISA Move In8 Function

VISA Move In8 / Move In16 / Move In32

VISA Move In16 Function

VISA Move In8 / Move In16 / Move In32

VISA Move In32 Function

VISA Move In8 / Move In16 / Move In32

VISA Move Out8 Function

VISA Move Out8 / Move Out16 / Move Out32

VISA Move Out16 Function

VISA Move Out8 / Move Out16 / Move Out32

VISA Move Out32 Function

VISA Move Out8 / Move Out16 / Move Out32

VISA Find Resource Function

VISA Find Resource

VISA Open Function

VISA Open

VISA Close Function

VISA Close

VISA Read Function

VISA Read

VISA Read STB Function

VISA Read STB

VISA Write Function

VISA Write

VISA Clear Function

VISA Clear

VISA Trigger Function

VISA Trigger

VISA Lock Function

VISA Lock

VISA Unlock Function

VISA Unlock

VISA Status Description Function

VISA Status Description

VISA Attribute Node Function

VISA Overview