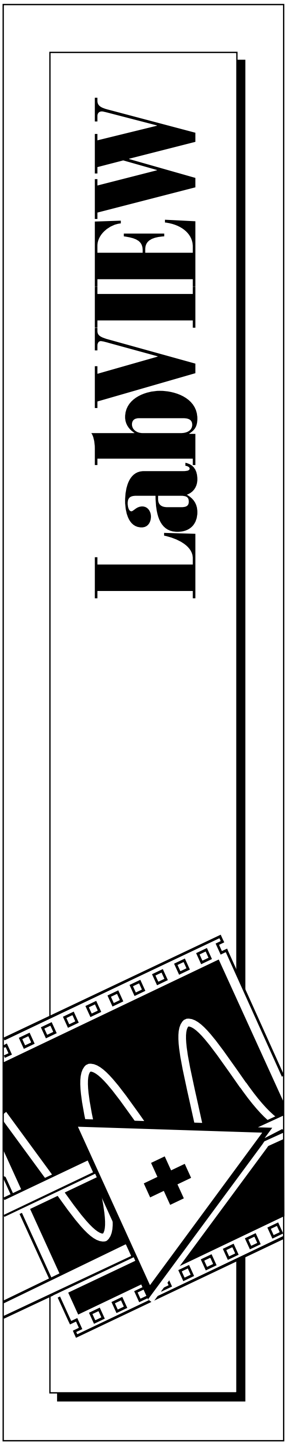


LabVIEW
Function and VI
Reference Manual

May 1997 EditionPart
Number 321526A-01

Internet Support

`support@natinst.com`
E-mail: `info@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `http://www.natinst.com`

Bulletin Board Support

BBS United States: (512) 794-5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support

(512) 418-1111

Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678

International Offices

Australia 02 9874 4100, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 527 2321, France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 5734815, Italy 06 5729961, Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 31 348 43 34 66, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway   Austin, TX 78730-5039   Tel: (512) 794-0100

Important Information

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

About This Manual

Chapter 1
Introduction to the LabVIEW Functions and VIs

Section One: G Functions and VIs

Chapter 2
G Function and VI Reference Overview

Chapter 3
Structures

Chapter 4
Numeric Functions

Chapter 5
Boolean Functions

Chapter 6
String Functions

Chapter 7
Array Functions

Chapter 8
Cluster Functions

Chapter 9
Comparison Functions

Chapter 10
Time, Dialog, and Error Functions

Chapter 11
File Functions

Chapter 12
Advanced Functions

Section Two: Data Acquisition VIs

Chapter 13
Introduction to the LabVIEW Data Acquisition VIs

Chapter 14
Easy Analog Input VIs

Chapter 15
Intermediate Analog Input VIs

Chapter 16
Analog Input Utility VIs

Chapter 17
Advanced Analog Input VIs

Chapter 18
Easy Analog Output VIs

Chapter 19
Intermediate Analog Output VIs

Chapter 20
Analog Output Utility VIs

Chapter 21
Advanced Analog Output VIs

Chapter 22
Easy Digital I/O VIs

Chapter 23
Intermediate Digital I/O VIs

Chapter 24
Advanced Digital I/O VIs

Chapter 25
Easy Counter VIs

Chapter 26
Intermediate Counter VIs

Chapter 27
Advanced Counter VIs

Chapter 28
Calibration and Configuration VIs

Chapter 29
Signal Conditioning VIs

Section Three: Instrument I/O Functions and VIs

Chapter 30
Introduction to LabVIEW Instrument Driver VIs

Chapter 31
LabVIEW Instrument Driver Models

Chapter 32
LabVIEW Instrument Driver Development

Chapter 33
Instrument Driver Template VIs

Chapter 34
VISA Library Reference

Chapter 35
Traditional GPIB Functions

Chapter 36
GPIB 488.2 Functions

Chapter 37
Serial Port VIs

Section Four: Analysis VIs

Chapter 38
Introduction to Analysis in LabVIEW

Chapter 39
Analysis Signal Generation VIs

Chapter 40
Analysis Digital Signal Processing VIs

Chapter 41
Analysis Measurement VIs

Chapter 42 Analysis Filter VIs

Chapter 43 Analysis Window VIs

Chapter 44 Analysis Curve-Fitting VIs

Chapter 45 Analysis Probability and Statistics VIs

Chapter 46
Analysis Linear Algebra VIs

Chapter 47
Analysis Array Operation VIs

Chapter 48
Analysis Additional Numerical Method VIs

Section Five: Communication VIs and Functions

Chapter 49
Introduction to LabVIEW Communication VIs and Functions

Chapter 50
TCP VIs

Chapter 51
UDP VIs

Chapter 52
DDE VIs

Chapter 53
OLE Automation VIs

Chapter 54
AppleEvent VIs

Chapter 55
Program to Program Communication VIs

Appendix A
DAQ Hardware Capabilities

Appendix B
Multiline Interface Messages

Appendix C
Operation of the GPIB

Appendix D
References

Appendix E
Customer Communication

Index

Figures

Tables

About This Manual

The *LabVIEW Function and VI Reference Manual* contains descriptions of all virtual instruments (VIs) available for LabVIEW, including the following:

- G functions and VIs
- VIs that support the devices for data acquisition
- VIs for GPIB, VXIbus, and serial port I/O operation
- digital signal processing, filtering, and numerical and statistical VIs
- networking and interapplication communications VIs

This manual is a supplement to the *LabVIEW User Manual* and assumes that you are familiar with that material. You should also know how to operate LabVIEW, and your computer and operating system.

This manual provides an overview of each function and VI available in LabVIEW. However, for more specific information regarding each function and VI (e.g. for specific parameter information), refer to the LabVIEW Online Reference, which you can access by selecting **Online Reference** from the LabVIEW **Help** menu, or Help, which you access by selecting **Show Help** from the LabVIEW **Help** menu.

Organization of the Product User Manual

This manual covers five subject areas G Functions, Data Acquisition VIs, Instrument I/O VIs, Analysis VIs, and Communications VIs. Chapter 1 introduces the LabVIEW Functions and VIs, which comprise the sections in this manual.

- Section 1, G Functions and VIs, includes Chapters 2 through 12, which describe the functions unique to the G programming language.
- Section 2, Data Acquisition VIs, includes Chapters 13 through 29, which describe the Data Acquisition (DAQ) VIs.

- Section 3, Instrument I/O Functions and VIs, includes Chapters 30 through 37, which describe the Instrument I/O VIs and functions.
- Section 4, Analysis VIs, includes Chapters 38 through 48, which describe the Analysis VIs.
- Section 5, Communications VI and Functions, includes Chapters 49 through 55, which describe the Communication VIs.

In addition, this manual includes the following appendices and index:

- Appendix A, *DAQ Hardware Capabilities*, includes tables that summarize the analog and digital I/O capabilities of National Instruments data acquisition devices.
- Appendix B, *Multiline Interface Messages*, lists commands that IEEE 488 defines.
- Appendix C, *Operation of the GPIB*, describes basic concepts you need to understand to operate the GPIB.
- Appendix D, *References*, lists the reference material used to produce the Analysis VIs described in this manual.
- Appendix E, *Customer Communication*, contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation.
- The *Index* contains an alphabetical list of VIs described in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

< > Angle brackets enclose the name of a key on the keyboard (for example, <option>). Angle brackets containing numbers separated by an ellipsis represent a range of values associated with a bit or signal name (for example, DBIO<3…0>).

[ ] Square brackets enclose optional items (for example, [`response`]).

- A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys–for example, <Control-Alt-Delete>.

» The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options» Substitute Fonts** directs you to pull down the **File** menu, select the

Page Setup item, select **Options**, and finally select the **Substitute Fonts** options from the last dialog box.

♦ The ♦ symbol indicates that the text following it applies only to a specific product, a specific operating system, or a specific software version.

bold Bold text denotes the names of menus, menu items, parameters, dialog box, dialog box buttons or options, icons, windows, Windows 95 tabs, or LEDs.

bold italic Bold italic text denotes a note, caution, or warning.

`bold monospace` Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

CTRL Key names are in all capital letters.

italic Italic text denotes emphasis, a cross reference, or an introduction to a key concept. This font also denotes text from which you supply the appropriate word or value, as in Windows 3.*x*.

`italic monospace` Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.

`monospace` Text in this font denotes text or characters that should literally enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and for statements and comments taken from programs.

paths Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.

The *Glossary* lists abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms.

Related Documentation

You might find the following documents helpful as you read this manual:

- *LabVIEW User Manual*
- *LabVIEW Error Codes*
- *LabVIEW Getting Started Card*
- *LabVIEW QuickStart Guide*

- *LabVIEW Release Notes*
- *LabVIEW Upgrade Notes*
- *G Quick Reference Card*

Related Online Documentation

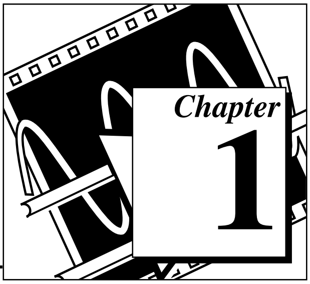
The following related documents are available through the LabVIEW Online Reference, which you access by selecting **Help»OnlineReference**.

- *Communications Common Questions*
- *LabVIEW Glossary*

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix E, *Customer Communication*, at the end of this manual.

Introduction to the LabVIEW Functions and VIs



This chapter contains basic information about the functions and virtual instruments (VIs) that are available with LabVIEW.

LabVIEW includes collections of VIs that work with your G programming language, data acquisition (DAQ) hardware devices, instrument input and output devices, analysis instruments, and communication devices.

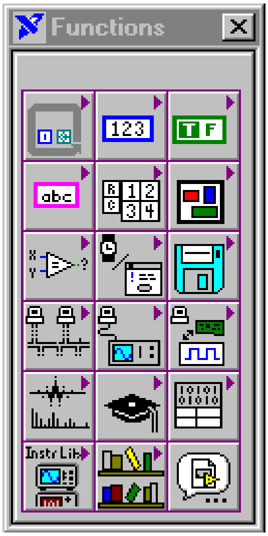
Locating the G Functions and VIs

You can find the G functions and VIs on the **Functions** palette. To access the **Functions** palette, access a block diagram in LabVIEW. When you put your cursor over each of the icons in the **Functions** palette, LabVIEW displays the name of the icon palette.

Functions are elementary nodes in the G programming language. They are analogous to operators or library functions in conventional languages. Functions are not VIs and therefore do not have front panels or block diagrams. When compiled, functions generate inline machine code.

You select functions from the **Functions** palette, in the block diagram. When the block diagram window is active, select **Windows»Show Functions Palette**. You also can access the

Functions palette by popping up on the area in the block diagram window where you want to place the function.



Structures, Numeric Boolean

String, Array, Cluster

Comparison, Time & Dialog, File I/O

Communication, Instrument I/O, DAQ

Analysis, Tutorial, and Advanced

Instrument Driver, User Libraries, Select a VI

Many Function palette chapters include information about function examples.

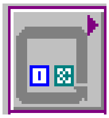
The paths for these examples for LabVIEW begin with `examples\`.

Function and VI Overviews

The following functions and VIs are available.

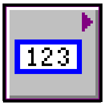
Structures

G Structures include While Loop, For Loop, Case and Sequence structures. This palette also contains the global and local variable nodes.



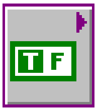
Numeric Functions

Numeric functions perform arithmetic operations, conversions, trigonometric, logarithmic, and complex mathematical functions. This palette also contains additional numeric constants, such as Pi.



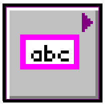
Boolean Functions

Boolean functions perform Boolean and logical operations.



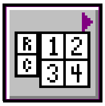
String Functions

String functions manipulate strings and convert numbers to and from strings. This palette also includes the subpalettes Additional String To Number Functions and String Conversion Functions.



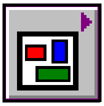
Array Functions

Array functions assemble, disassemble, and process arrays.



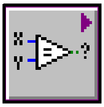
Cluster Functions

Use Cluster functions to assemble, access, and change elements in a cluster.



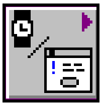
Comparison Functions

Comparison functions compare data (greater than, less than, and so on) and operations that are based on a comparison, such as finding the minimum and maximum ranges for two values.



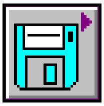
Time and Dialog Functions

Time and Dialog functions can be used to manipulate time functions and display dialog boxes. This palette also includes the functions that perform error handling.



File I/O Functions

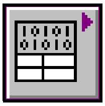
File I/O functions manipulate files and directories. This palette also contains the subpalettes Advanced File Functions, Binary File VIs, and File Constants.



Advanced Functions

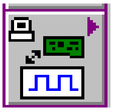
Advanced functions are functions that do not fit into any other category. The Code Interface Node is an example of an advanced function. The

Advanced Functions palette also contains Help Window functions, VI Control VIs, Data Manipulation functions, and Occurrences functions.



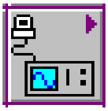
DAQ

DAQ VIs acquire and generate real-time analog and digital data as well as perform counting operations. See Chapter 13, *Introduction to the LabVIEW Data Acquisition VIs*, for more information.



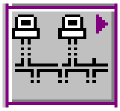
Instrument I/O

Instrument I/O VIs communicate with instruments using GPIB, VISA, or serial communication. See Chapter 30, *Introduction to LabVIEW Instrument Driver VIs*, for more information.



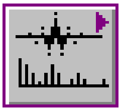
Communication

Communication VIs network to other applications using TCP/IP, DDE, OLE, Apple Events, PPC, or UDP. See Chapter 49, *Introduction to LabVIEW Communication VIs and Functions*, for more information.



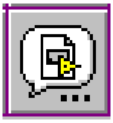
Analysis VIs

Analysis VIs perform measurement, signal generation, digital signal processing, filtering, windowing, probability and statistics, curve fitting, linear algebra, array operations, and VIs which perform additional numerical methods. See Chapter 38, *Introduction to Analysis in LabVIEW*, for more information.



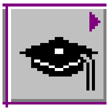
Select A VI...

When you select **Functions»Select a VI...**, LabVIEW displays a file dialog box. From there, you can select any VI and place it on a diagram.



Tutorial

Selecting **Functions»Tutorial** accesses the Tutorial VIs. You call these VIs while working through the *LabVIEW Tutorial Manual.*

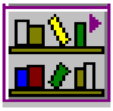


Instrument Driver Library

Instrument drivers are a set of VIs for GPIB, VISA, Serial, and CAMAC instruments. National Instruments, as well as other vendors, distribute these instrument drivers. Any drivers you place in the `instr.lib` appear in the palette.

User Library

This palette automatically includes any VIs in your `user.lib` directory, making it more convenient to gain access to commonly used sub-VIs you have written.



Chapter 2

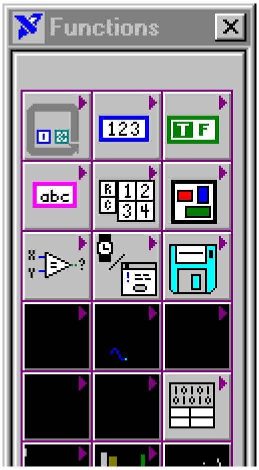
G Function and VI Reference Overview

This chapter introduces the G Functions and VIs, descriptions of which comprise Chapter 3 through Chapter 12.

Functions are elementary nodes in the G programming language. They are analogous to operators or library functions in conventional languages. Functions are not VIs and therefore do not have front panels or block diagrams. When compiled, functions generate inline machine code.

VIs are “virtual instruments,” so called because they model the appearance functions of a physical instrument.

You select G Functions from the **Functions** palette, in the block diagram. When the block diagram window is active, you can display the **Functions** palette by selecting **Windows»Show Functions Palette**. You also can access the **Functions** palette by popping up on the area in the block diagram window where you want to place the function.



G Functions and VIs

Many Functions palette chapters include information about function examples. The paths for these examples for LabVIEW begin `examples\`.

G Functions Overview

For brief descriptions of each of the 10 G Function and VI palettes available refer to Chapter 1, *Introduction to LabVIEW Functions and VIs*.

Introduction to Polymorphism

The following sections provide some general information about Polymorphism in G functions.

Polymorphism

Polymorphism is the ability of a function to adjust to input data of different types or representations. Most functions are polymorphic. VIs are not polymorphic. All functions that take numeric input can accept any numeric representation (except some functions that do not accept complex numbers).

Functions are polymorphic to varying degrees; none, some, or all of their inputs may be polymorphic. Some function inputs accept numbers or Boolean values. Some accept numbers or strings. Some accept not only scalar numbers but also arrays of numbers, clusters of numbers, arrays of clusters of numbers, and so on. Some accept only one-dimensional arrays although the array elements may be of any type. Some functions accept all types of data, including complex numbers.

Unit Polymorphism

If you want to create a VI that computes the root, mean square value of a waveform, you have to define the unit associated with the waveform. You would need a separate VI for voltage waveforms, current waveforms, temperature waveforms, and so on. LabVIEW has polymorphic unit capability so that one VI can perform the same calculation, regardless of the units received by the inputs.

You create a polymorphic unit by entering `$`*x*, where *x* is a number (for example, `$1`). You can think of this as a placeholder for the actual unit. When LabVIEW calls the VI, the program substitutes the units you pass in for all occurrences of `$`*x* in that VI.

LabVIEW treats a polymorphic unit as a unique unit. You cannot convert a polymorphic unit to any other unit, and polymorphic units propagate throughout the diagram, just as other units do. When the unit connects to an indicator that also has the abbreviation `$1`, the units match and the VI can then compile.

You can use `$1` in combinations just like any other unit. For example, if the input is multiplied by 3 seconds and then wired to an indicator, the indicator must be `$1 s` units. If the indicator has different units, the block diagram shows a bad wire. If you need to use more than one polymorphic unit, you can use the abbreviations `$2`, `$3`, and so on.

A call to a subVI containing polymorphic units computes output units based on the units received by its inputs. For example, suppose you create a VI that has two inputs with the polymorphic units `$1` and `$2` that creates an output in the form `$1 $2 / s`. If a call to the VI receives inputs with the unit m/s to the `$1` input and kg to the `$2` input, LabVIEW computes the output unit as `kg m / s^2`.

Suppose a different VI has two inputs of `$1` and `$1/s`, and computes an output of `$1^2`. If a call to this VI receives inputs of m/s to the `$1` input and `m/s^2` to the `$1/s` input, LabVIEW computes the output unit as `m^2 / s^2`. If this VI receives inputs of m to the `$1` input and kg to the `$1/s` input, however, LabVIEW declares one of the inputs as a unit conflict and computes (if possible) the output from the other input.

A polymorphic VI can have a polymorphic subVI because LabVIEW keeps the respective units distinct.

Numeric Conversion

You can convert any numeric representation to any other numeric representation. When you wire two or more numeric inputs of different representations to a function, the function usually returns output in the larger or wider format. The functions coerce the smaller representations to the widest representation before execution.

Some functions, such as Divide, Sine, and Cosine, always produce floating-point output. If you wire integers to their inputs, these functions convert the integers to double-precision, floating-point numbers before performing the calculation.

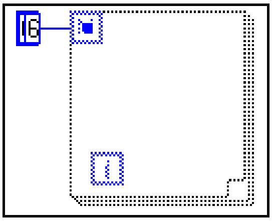
For floating-point, scalar quantities, it is usually best to use double-precision, floating-point numbers. Single-precision, floating-point numbers save little memory, little or no time, and overflow much more easily. You should only use extended-precision, floating-point numbers when necessary. The performance and precision of extended-precision arithmetic varies among the platforms.

For integers, it is usually best to use a long integer.

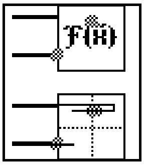
If you wire an output to a destination that has a different numeric representation from the source, LabVIEW converts the data according to the following rules:

- Signed or unsigned integer to floating-point number—Conversion is exact, except for long integers to single-precision, floating-point numbers. In this case, LabVIEW reduces the precision from 32 bits to 24 bits.
- Floating-point number to signed or unsigned integer—LabVIEW moves out-of-range values to the integer's minimum or maximum value. In most integer objects, such as the iteration terminal of a For Loop, LabVIEW rounds floating-point numbers. LabVIEW rounds a fractional part of 0.5 to the nearest even integer—for example, LabVIEW rounds 6.5 to 6 rather than 7.
- Integer to integer—LabVIEW does not move out-of-range values to the integer’s minimum or maximum value. If the source is smaller than the destination, LabVIEW extends the sign of a signed source and places zeros in the extra bits of an unsigned source. If the source is larger than the destination, LabVIEW copies only the low order bits of the value.

On the block diagram, LabVIEW places a *coercion dot* on the border of a terminal where the conversion takes place to indicate that automatic numeric conversion occurred, as in the following example.



Because VIs and functions can have many terminals, a coercion dot can appear inside an icon if the wire crosses an internal terminal boundary before it leaves the icon/connector, as the following illustration shows.



Moving a wired icon stretches the wire. Coercion dots can cause a VI to use more memory and time. You should try to keep data types consistent in your VIs. For more information on coercion dots, see Chapter 8, *Customizing Your LabVIEW Environment*, in the *LabVIEW User Manual*.

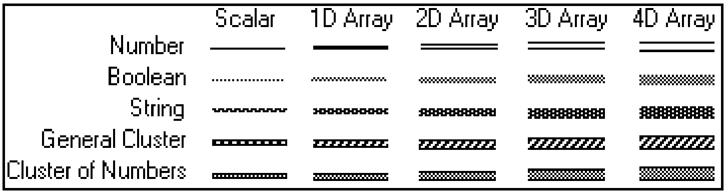
Overflow and Underflow

LabVIEW does not check for overflow or underflow conditions on integer values. Overflow and underflow for floating-point numbers is in accordance with IEEE 488 Standard 754 for binary, floating-point arithmetic.

Floating-point operations propagate not-a-number (NaN) and +/-Inf faithfully. When you explicitly or implicitly convert NaN or +/-Inf to an integer or Boolean value, however, you get a value that looks reasonable, but is meaningless. For example, dividing by zero produces +/-Inf, but converting that value to a word integer gives the value 32,768, which is the largest value that can be represented in the destination format.

Wire Styles

The wire style represents the data type for each terminal, as the following table shows. Polymorphic functions show the wire style for the most commonly used data type.

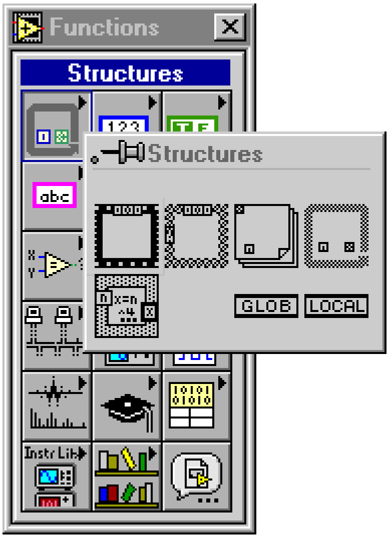

Scalar
1D Array
2D Array
3D Array
4D Array
Number
Boolean
String
General Cluster
Cluster of Numbers

Chapter 3

Structures

This chapter describes the Structures available through LabVIEW.

To access the **Structures** palette, select **Functions»Structures**. The following illustration shows the options that are available on the **Structures** palette.



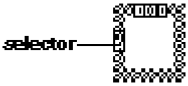
See `examples\general\structs.llb` for examples of how these structures are used in LabVIEW.

Structures Overview

The following Structures are available in LabVIEW.

Case Structure

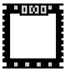
Has one or more subdiagrams, or *cases*, exactly one of which executes when the structure executes. Whether or not it executes depends on the value of the Boolean or numeric scalar you wire to the external side of the terminal or *selector*.



For more information on how to use the Case structure in LabVIEW, see Chapter 19, *Structures*, in the *LabVIEW User Manual*.

Sequence Structure

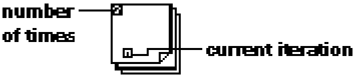
Consists of one or more subdiagrams, or *frames*, that execute sequentially. As an option, you can add sequence locals that allow you to pass information from one frame to subsequent frames by popping up on the edge of the structure.



For more information on how to use the Sequence structure in LabVIEW, see Chapter 19, *Structures*, in the *LabVIEW User Manual*.

For Loop

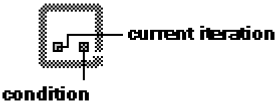
Executes its subdiagram count times, where the count equals the value contained in the count terminal. As an option, you can add shift registers so you can pass information from one iteration to the next by popping up on the edge of the structure.



For more information on how to use For Loop in LabVIEW, see Chapter 19, *Structures*, in the *LabVIEW User Manual*.

While Loop

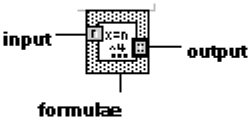
Executes its subdiagram until a Boolean value you wire to the *conditional terminal* is FALSE. As an option, you can add shift registers so you can pass information from one iteration to the next by popping up on the edge of the structure.



For more information on how to use While Loop in LabVIEW, see Chapter 19, *Structures*, in the *LabVIEW User Manual*.

Formula Node

Executes mathematical formulas on the block diagram.



For more information on the Formula Node, see Chapter 20, *The Formula Node*, in the *LabVIEW User Manual*.

Global Variable

A built-in LabVIEW object that you define by creating a special kind of VI, with front panel controls that define the datatype of the global variable.

For more information on the global variable, see Chapter 22, *Global and Local Variables*, in the *LabVIEW User Manual*.

Local Variable

Lets you read or write one of the controls or indicators on the front panel of your VI. Writing to a local variable has the same result as passing data to a terminal, except that you can write to it even though it is a control, or read from it even though it is an indicator.

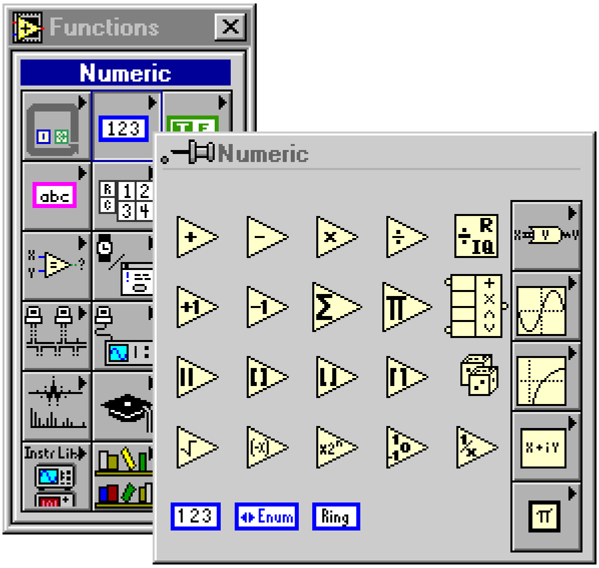
For more information on the local variable, see Chapter 22, *Global and Local Variables*, in the *LabVIEW User Manual*.

Chapter 4

Numeric Functions

This chapter describes the functions that perform arithmetic operations, complex, conversion, logarithmic, and trigonometric operations. It also describes the commonly used constants like the numeric constant, enumerated constant, and ring constant as well additional numeric constants.

To access the **Numeric** palette, select **Functions»Numeric**. The following illustration shows the options that are available on the **Numeric** palette.



The **Numeric** palette includes the following subpalettes:

- Additional Numeric Constants
- Complex
- Conversion
- Logarithmic
- Trigonometric

For examples of some of the arithmetic functions, see `examples\general\structs.llb`.

Polymorphism for Numeric Functions

The arithmetic functions accept numeric input data. With some exceptions noted in the function descriptions, the output has the same numeric representation as the input, or if the inputs have different representations, the output is the wider of the inputs.

The arithmetic functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on. A formal and recursive definition of the allowable input type is as follows:

> *Numeric type* = numeric scalar || array [*numeric type*] || cluster [*numeric types*]
>
> The numeric scalars can be a floating-point, integer or complex, number. G does not allow you to use arrays of arrays.

Arrays can have any number of dimensions of any size. Clusters can have any number of elements. For functions with one input, the functions operate on each element of the structure.

For functions with two inputs, you can use the following input combinations:

- *Similar*—both inputs have the same structure, and the output has the same structure as the inputs.
- *One scalar*—one input is a numeric scalar, the other is an array or cluster, and the output is an array or cluster.
- *Array of*—one input is a numeric array, the other is the numeric type itself, and the output is an array.

For similar inputs, G performs the function on the respective elements of the structures. For example, G can add two arrays element-by-element. Both arrays must have the same dimensionality. You can add arrays with differing numbers of elements; the output of such an addition has the same number of elements as the smallest input. Clusters also must have the same number of elements, and the respective elements must have the same structure.

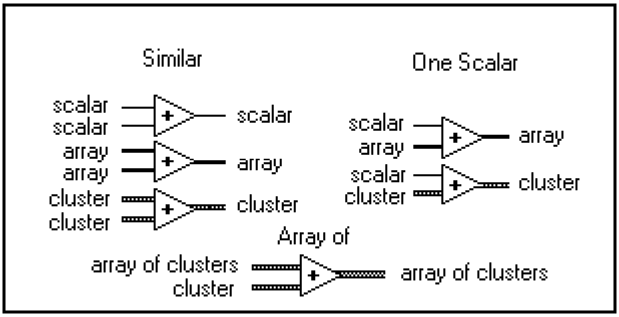
☞ **Note:** ***You cannot use the multiply function to do matrix multiplication. If you use the multiply function with two matrices, G takes the first number in the***

first row of the first matrix, multiplies it by the first number in the first row of the second matrix, and so on.

For operations involving a scalar and an array or cluster, G performs the function on the scalar and the respective elements of the structure. For example, G can subtract a number from all elements of an array, regardless of the dimensionality of the array.

For operations that involve a numeric type and an array of that type, G performs the function on each array element. For example, a graph is an array of points, and a point is a cluster of two numeric types, *x* and *y*. To offset a graph by 5 units in the *x* direction and 8 units in the *y* direction, you can add a point, (5, 8), to the graph.

See the Polymorphic Combinations example below illustrates some of the possible polymorphic combinations of the Add function.



Polymorphism for Trig Functions

The trigonometric functions accept numeric input data. If the input is an integer, the output is a double-precision, floating-point number. Otherwise, the output has the same numeric representation as the input.

These functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on.

Polymorphism for Logarithmic Functions

The logarithmic functions accept numeric input data. If the input is an integer, the output is a double-precision, floating-point number. Otherwise, the output has the same numeric representation as the input.

These functions work on numbers, arrays of numbers, clusters of numbers, arrays of clusters of numbers, complex numbers, and so on.

Polymorphism for Conversion Functions

All the conversion functions except Byte Array to String, String to Byte Array, Convert Unit, and Cast Unit Bases are polymorphic. That is, the polymorphic functions work on scalar values, arrays of scalars, clusters of scalars, arrays of clusters of scalars, and so on. The output has the same numeric representation as the input but with the new type.

Polymorphism for Complex Functions

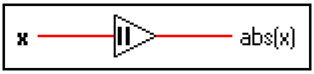
The complex functions work on scalar values, arrays of scalars, clusters of scalars, arrays of clusters of scalars, and so on. The output has the same composition as the input but with the new type.

Arithmetic Function Descriptions

The following functions are available.

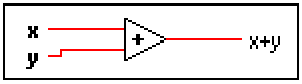
Absolute Value

Returns the absolute value of the input.



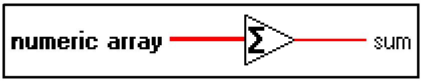
Add

Computes the sum of the inputs.



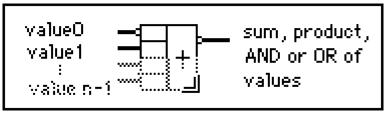
Add Array Elements

Returns the sum of all the elements in **numeric array.**



Compound Arithmetic

Performs arithmetic on two or more numeric, cluster, or Boolean inputs.



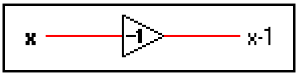
You select the operation (multiply, AND, or OR) by popping up on the function and selecting **Change Mode**.

You can invert the inputs or the output of this function by popping up on the individual terminals, and selecting **Invert**. For Add, select **Invert** to negate an input or the output. For Multiply, select **Invert** to use the reciprocal of an input or to produce the reciprocal of the output. For AND or OR, select **Invert** to logically negate an input or the output.

☞ **Note:** ***You add inputs to this node by popping up on an input and selecting Add Input or by placing the Positioning tool in the lower left or right corner of the node and dragging it.***

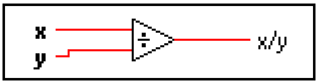
Decrement

Subtracts 1 from the input value.



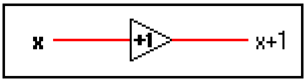
Divide

Computes the quotient of the inputs.



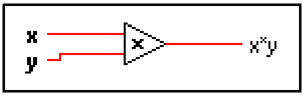
Increment

Adds 1 to the input value.



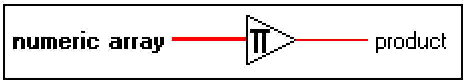
Multiply

Returns the product of the inputs.



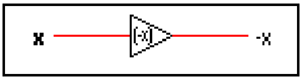
Multiply Array Elements

Returns the **product** of all the elements in **numeric array.**



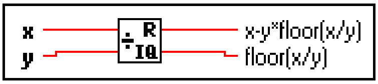
Negate

Negates the input value.



Quotient & Remainder

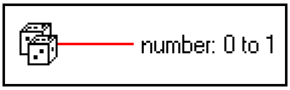
Computes the integer quotient and the remainder of the inputs.



With integer input values for **y** of zero, the quotient is zero and the remainder is the dividend **x**. For floating point inputs, if y is zero, the quotient is infinity and the remainder defaults to NaN.

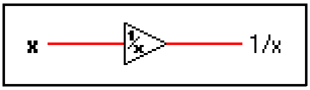
Random Number (0–1)

Produces a double-precision floating-point number between 0 and 1 exclusive, or not including 0 and 1. The distribution is uniform.



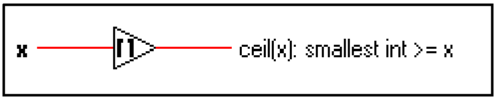
Reciprocal

Divides 1 by the input value.



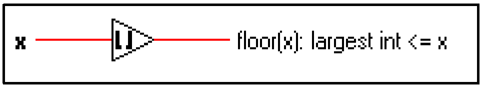
Round To +Infinity

Rounds the input to the next highest integer. For example, if the input is 3.1, the result is 4. If the input is –3.1, the result is –3.



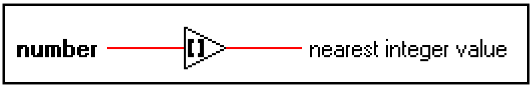
Round To –Infinity

Rounds the input to the next lowest integer. For example, if the input is 3.8, the result is 3. If the input is –3.8, the result is –4.



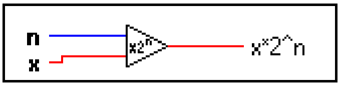
Round To Nearest

Rounds the input to the nearest integer. If the value of the input is midway between two integers (for example, 1.5 or 2.5), the function returns the nearest even integer (2).



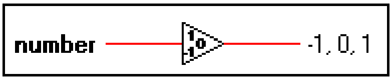
Scale By Power Of 2

Multiplies one input (**x**) by 2 raised to the power of the other input (**n**). If **n** is floating-point, this function rounds **n** prior to scaling **x** (0.5 rounds to 0; 0.51 rounds to 1). If **x** is an integer, this function is the equivalent of an arithmetic shift.



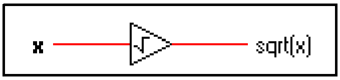
Sign

Returns 1 if the input value is greater than 0, returns 0 if the input value is equal to 0, and returns –1 if the input value is less than 0. Other programming languages typically call this function the `signum` or `sgn` function.



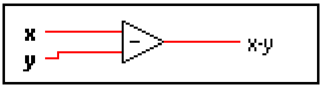
Square Root

Computes the square root of the input value. If **x** is negative, the square root is not a number (NaN) unless **x** is complex.



Subtract

Computes the difference of the inputs.



User Definable Arithmetic Constants

You can define the following constants.

Numeric Constant

Use this to supply a constant numeric value to the block diagram. Set this value by clicking inside the constant with the Operating tool and typing in a value. You can change the data format and representation.

The value of the numeric constant cannot be changed while the VI executes. You can assign a label to this constant.

Enumerated Constant

Enumerated values associate unsigned integers to strings. If you display a value from an enumerated constant the string displays instead of the number associated with it. If you need a set of strings that will not change, then use this constant. Set the value by clicking inside the constant with the Operating Tool. Set the string with the Labeling Tool and enter the string. To add another item, click on the constant and choose **Add Item Before** or **Add Item After**.

The value of the enumerated constant cannot be changed while the VI executes. You can assign a label to this constant.

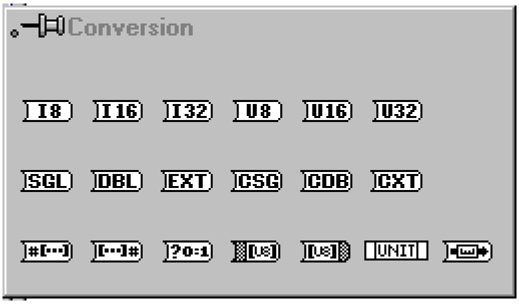
Ring Constant

Rings can be used to associate unsigned integers to strings. If you display a value from a ring constant the number displays instead of the string associated with it. If you need a set of strings that will not change, then use this constant. Set the value by clicking inside the constant with the Operating Tool. Set the string with the Labeling Tool and enter the string. To add another item, pop up on the constant and choose **Add Item Before** or **Add Item After**.

The value of the ring constant cannot be changed while the VI executes. You can assign a label to this constant.

Conversion Functions Descriptions

The following illustration shows the options that are available on the **Conversion** subpalette.



The following functions convert a numeric input into a specific representation:

- To Byte Integer
- To Double Precision Complex
- To Double Precision Float
- To Extended Complex
- To Extended Precision Float
- To Long Integer
- To Single Precision Complex
- To Single Precision Float
- To Unsigned Byte Integer

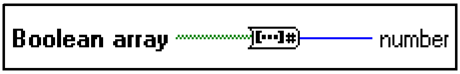
- To Unsigned Word Integer
- To Unsigned Long Integer
- To Word Integer

When these functions convert a floating-point number to an integer, they round the output to the nearest integer, or the nearest even integer if the fractional part is 0.5. If the result is out of range for the integer, these functions return the minimum or maximum value for the integer type. When these functions convert an integer to a smaller integer, they copy the least significant bits without checking for overflow. When they convert an integer to a larger integer, they extend the sign of a signed integer and pad an unsigned integer with zeros.

Use caution when you convert numbers to smaller representations, particularly when converting integers, because the G conversion routines do not check for overflow.

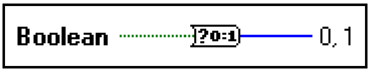
Boolean Array To Number

Converts **Boolean array** to an unsigned long integer by interpreting it as the two's complement representation of an integer with the 0th element of the array being the least significant bit.



Boolean To (0,1)

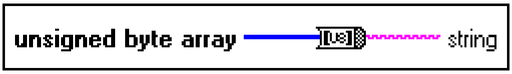
Converts a Boolean value to a word integer— 0 and 1 for the input values FALSE and TRUE, respectively.



Boolean can be a scalar, an array, or a cluster of Boolean values, an array of clusters of Boolean values, and so on. See the *Polymorphism for Boolean Functions* section in Chapter 5, *Boolean Functions*.

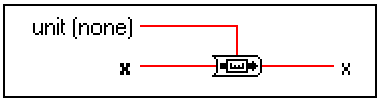
Byte Array To String

Converts an array of unsigned bytes into a string.



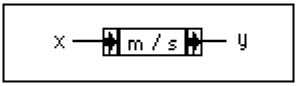
Cast Unit Bases

Changes the units associated with the input to the units associated with **unit** and returns the results at the output terminal. Use this function with extreme care. Because the **Cast Unit Bases** function works with bases, you must understand the conversion from an arbitrary unit to its bases before you can effectively use this function. This function can change base units, such as changing meters to grams.



Convert Unit

Converts a physical number (a number that has a unit) to a pure number (a number with no units) or a pure number to a physical number.



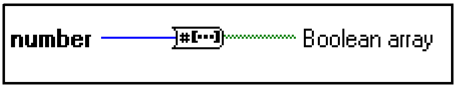
You can edit the string inside of the unit by highlighting the string with an Operating tool and then entering the text.

If the input is a pure number, the output receives the specified units. For example, given an input of 13 and a unit specification of seconds(s), the resulting value is 13 seconds.

If the input is a physical number, and **unit** is a compatible unit, the output is the input measured in the specified units. For example, if you specify 37 meters(m), and a **unit** is m, the result is 37 with no associated units. If **unit** is feet (ft), the result is 121.36 with no associated units.

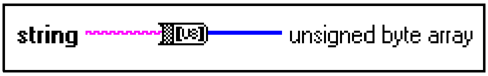
Number To Boolean Array

Converts an integer **number** to a Boolean array of 8, 16, or 32 elements, where the 0th element corresponds to the least significant bit (LSB) of the two’s complement representation of the integer.



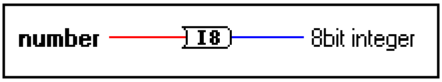
String To Byte Array

Converts a **string** into an array of unsigned bytes.



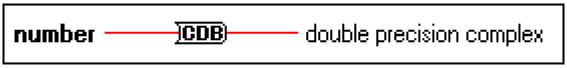
To Byte Integer

Converts **number** to an 8-bit integer in the range –128 to 127.



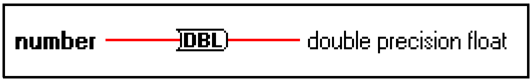
To Double Precision Complex

Converts a **number** to a double-precision complex number.



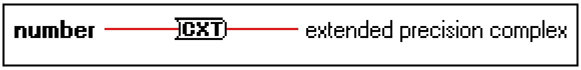
To Double Precision Float

Converts **number** to a double-precision floating-point number.



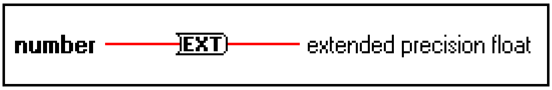
To Extend Precision Complex

Converts a **number** to an extended-precision complex number.



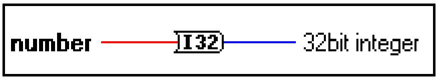
To Extended Precision Float

Converts **number** to an extended-precision floating-point number.



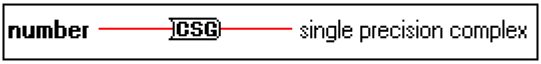
To Long Integer

Converts **number** to a 32-bit integer in the range -2^{31} to $2^{31}-1$



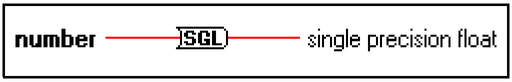
To Single Precision Complex

Coverts a **number** to a **single-precision complex number**.



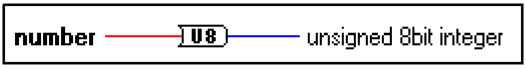
To Single Precision Float

Converts **number** to a single-precision floating-point number.



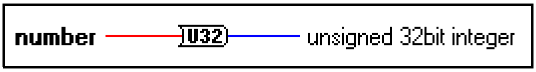
To Unsigned Byte Integer

Converts **number** to an 8-bit unsigned integer in the range 0 to 255.



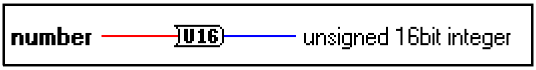
To Unsigned Long Integer

Converts **number** to a 32-bit unsigned integer in the range 0 to $2^{32} -1$.



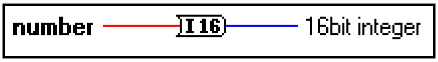
To Unsigned Word Integer

Converts **number** to a 16-bit unsigned integer in the range 0 to 65,535.



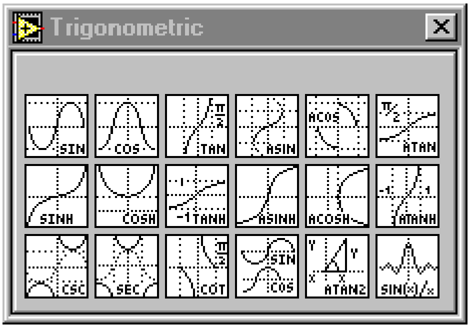
To Word Integer

Converts **number** to a 16-bit integer in the range –32,768 to 32,767.



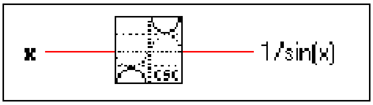
Trigonometric Functions Descriptions

The following illustration shows the options for the **Trigonometric** subpalette.



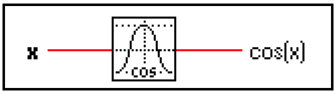
Cosecant

Computes the cosecant of **x**, where **x** is in radians. Cosecant is the reciprocal of sine.



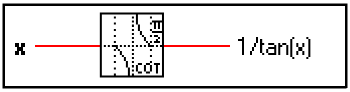
Cosine

Computes the cosine of **x**, where **x** is in radians.



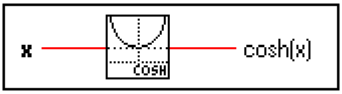
Cotangent

Computes the cotangent of **x**, where **x** is in radians. Cotangent is the reciprocal of tangent.



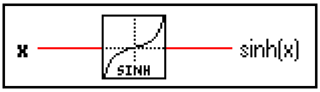
Hyperbolic Cosine

Computes the hyperbolic cosine of **x**, where **x** is in radians.



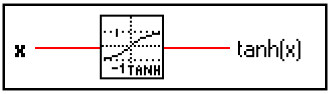
Hyperbolic Sine

Computes the hyperbolic sine of **x**, where **x** is in radians.



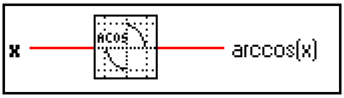
Hyperbolic Tangent

Computes the hyperbolic tangent of **x**, where **x** is in radians.



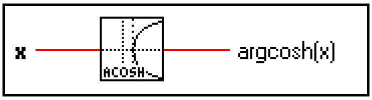
Inverse Cosine

Computes the arccosine of **x** in radians. If **x** is not complex and is less than –1 or greater than +1, the result is NaN.



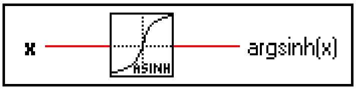
Inverse Hyperbolic Cosine

Computes the hyperbolic argcosine of **x** in radians. If **x** is not complex and is less than 1, the result is NaN.



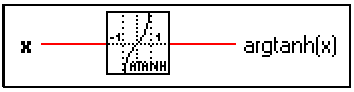
Inverse Hyperbolic Sine

Computes the hyperbolic argsine of **x** in radians.



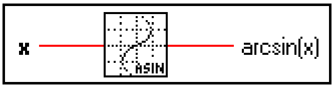
Inverse Hyperbolic Tangent

Computes the hyperbolic argtangent of **x** in radians. If **x** is not complex and is less than –1 or greater than 1, the result is NaN.



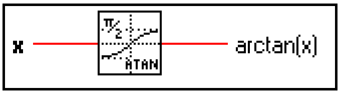
Inverse Sine

Computes the arcsine of **x** in radians. If **x** is not complex and is less than –1 or greater than +1, the result is NaN.



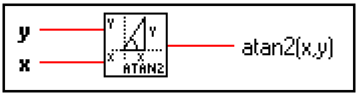
Inverse Tangent

Computes the arctangent of **x** in radians (which can be between –π/2 and π/2).



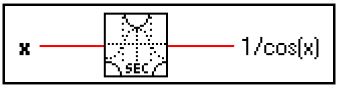
Inverse Tangent (2 Input)

Computes the arctangent of **y/x** in radians. This function can compute the arctangent for angles in any of the four quadrants of the x,y plane, whereas the Inverse Tangent function computes the arctangent in only two quadrants.



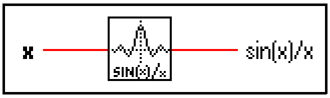
Secant

Computes the secant of **x**, where **x** is in radians.



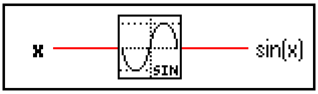
Sinc

Computes the sine of **x** divided by **x**, where **x** is in radians.



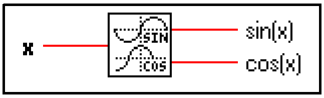
Sine

Computes the sine of **x**, where **x** is in radians.



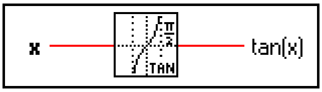
Sine & Cosine

Computes both the sine and cosine of **x**, where **x** is in radians. Use this function only when you need both results.



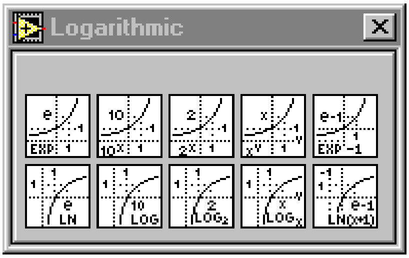
Tangent

Computes the tangent of **x**, where **x** is in radians.



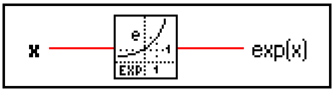
Logarithmic Functions Descriptions

The following illustration shows the options for the **Logarithmic** subpalette.



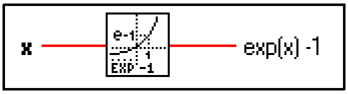
Exponential

Computes the value of e raised to the **x** power.



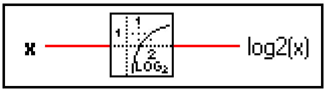
Exponential (Arg) –1

Computes 1 less than the value of e raised to the **x** power. When **x** is very small, this function is more accurate than using the Exponential function and then subtracting 1 from the output.



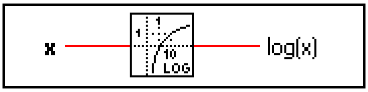
Logarithm Base 2

Computes the base 2 logarithm of **x**. If **x** is 0, **log2(x)** is –∞. If **x** is not complex and is less than 0, **log2(x)** is NaN.



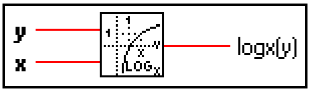
Logarithm Base 10

Computes the base 10 logarithm of **x**. If **x** is 0, **log(x)** is –∞. If **x** is not complex and is less than 0, **log(x)** is NaN.



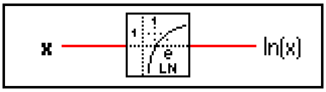
Logarithm Base X

Computes the base **x** logarithm of **y** (**x**>0, **y**>0). If **y** is 0, the output is –∞. When **x** and **y** are both not complex and **x** is less than or equal to 0, or **y** is less than 0, the output is NaN.



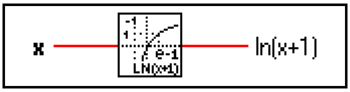
Natural Logarithm

Computes the natural base e logarithm of **x**, that is, the logarithm of **x**. If **x** is 0, **ln(x)** is –∞. If **x** is not complex and is less than 0, **ln(x)** is NaN.



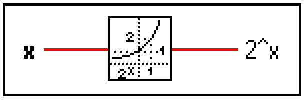
Natural Logarithm (Arg +1)

Computes the natural logarithm of (**x** + 1). When **x** is near 0, this function is more accurate than adding 1 to **x** and then using the Natural Logarithm function. If **x** is equal to –1, the result is –∞. If **x** is not complex and is less than –1, the result is NaN.



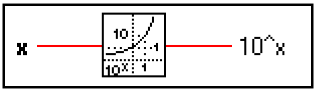
Power Of 2

Computes 2 raised to the **x** power.



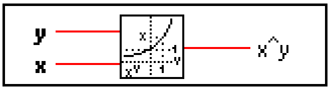
Power Of 10

Computes 10 raised to the **x** power.



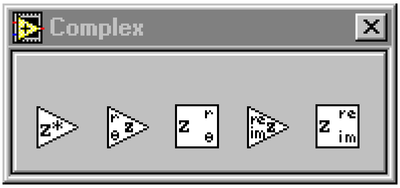
Power Of X

Computes **x** raised to the **y** power. If **x** is not complex, it must be greater than zero unless **y** is an integer value. Otherwise, the result is NaN. If **y** is zero, **x^y** is 1 for all values of **x**, including zero.



Complex Function Descriptions

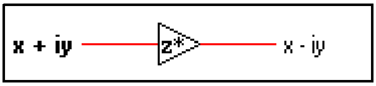
The following illustration displays the options available on the **Complex** subpalette.



The functions Polar To Complex and Re/Im To Complex create complex numbers from two values given in rectangular or polar notation, and the functions Complex To Polar and Complex To Re/Im break a complex number into its rectangular or polar components.

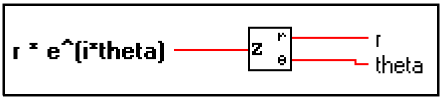
Complex Conjugate

Produces the complex conjugate of **x + iy**.



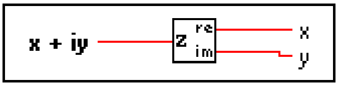
Complex To Polar

Breaks a complex number into its polar components.



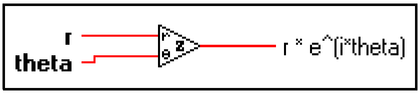
Complex To Re/Im

Breaks a complex number into its rectangular components.



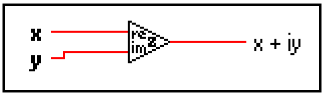
Polar To Complex

Creates a complex number from two values in polar notation.



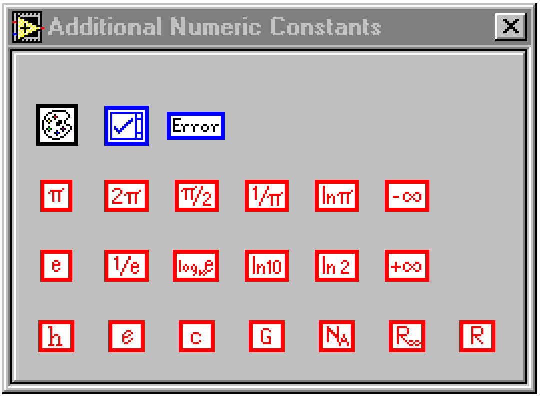
Re/Im To Complex

Creates a complex number from two values in rectangular notation.



Additional Numeric Constants Descriptions

The following illustration displays the options available on the **Additional Numeric Constants** subpalette.



Additional User Definable Constants

You can define the following constants.

Listbox Symbol Ring Constant

This ring constant assigns symbols to items in a listbox control. Typically, you wire this constant into the Item Symbols attribute.

Color Box Constant

Use this to supply a constant color value to the block diagram. Set this value by clicking on the constant with the Operating tool and choosing the desired color.

The value of the color box constant cannot be changed while the VI executes. You can assign a label to this constant.

Error Ring Constant

This constant is a predefined ring of errors specific to memory usage, networking, printing, and file I/O. Errors related to DAQ, GPIB, VISA, and Serial VIs and functions are not options in this ring.

Fixed Constants

The following constants are fixed.

Avogadro Constant (1/mol)

Returns the value 6.0220e23.

Base 10 Logarithm of e

Returns the value 0.43429448190325183.

Elementary Charge (c)

Returns the value 1.6021892e–19.

Gravitational Constant (Nm^2/kg^2)

Returns the value 6.6720e–11.

Molar Gas Constant (J/mol K)

Returns the value 8.31441.

e

Returns the value 2.7182818284590452e+0.

Natural Logarithm of Pi

Returns the value 1.14472988584940020.

Natural Logarithm of 2

Returns the value 0.69314718055994531.

Natural Logarithm of 10

Returns the value 2.30234095236904570.

Negative Infinity

Returns the value –∞.

Pi

Returns the value 3.14159265358979320.

Pi divided by 2

Returns the value 1.57079632679489660.

Pi multiplied by 2

Returns the value 6.28318530717958650.

Planck's Constant (J/Hz)

Returns the value 6.6262e–34.

Positive Infinity

Returns the value +∞.

Reciprocal of e

Returns the value 0.36787944117144232.

Reciprocal of Pi

Returns the value 0.31830988618379067.

Rydberg Constant (/m)

Returns the value 1.097373177e7.

Speed of Light in Vacuum (m/sec)

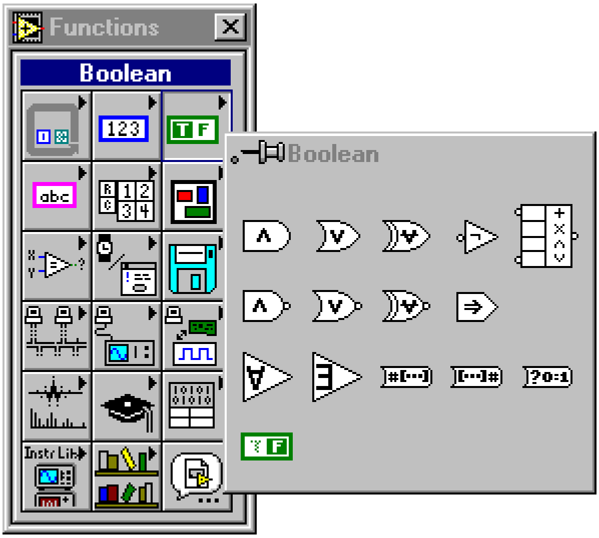
Returns the value 299,792,458.

Chapter 5

Boolean Functions

This chapter describes the functions that perform logical operations.

The following illustration shows the **Boolean** palette, which you access by selecting **Functions»Boolean**.



For examples of some of the Boolean functions, see `examples\general\structs.llb`.

Polymorphism for Boolean Functions

The logical functions take either Boolean or numeric input data. If the input is numeric, G performs a bit-wise operation. If the input is an integer, the output has the same representation. If the input is a floating-point number, G rounds it to a long integer, and the output is long integer.

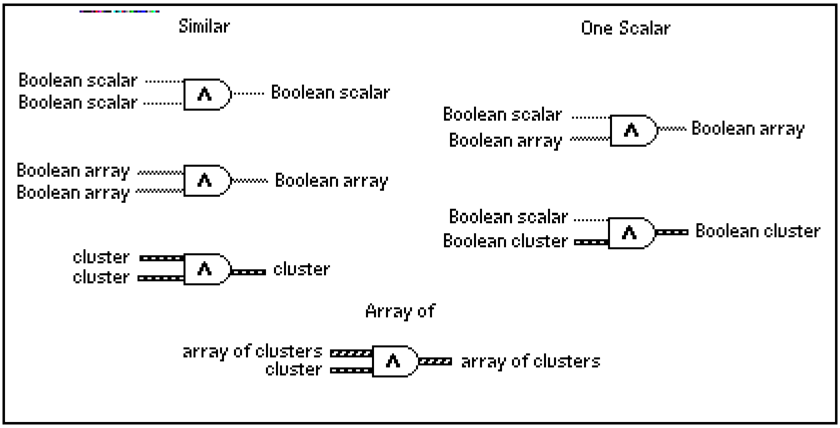
The logical functions work on arrays of numbers or Boolean values, clusters of numbers or Boolean values, arrays of clusters of numbers or

Boolean values, and so on. A formal and recursive definition of the allowable input type is as follows.

Logical type = Boolean scalar || numeric scalar || array [*logical type*] || cluster [*logical types*]

except that complex numbers and arrays of arrays are not allowed.

Logical functions with two inputs can have the same input combinations as the arithmetic functions. However, the logical functions have the further restriction that the base operations can only be between two Boolean values or two numbers. For example, you cannot have an AND between a Boolean value and a number. See the example below for an illustration of some combinations of Boolean values for the AND function.

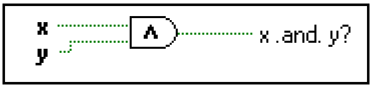


Boolean Function Descriptions

The following Boolean functions are available.

And

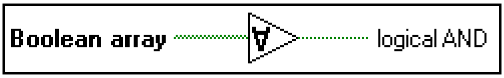
Computes the logical AND of the inputs.



☞ **Note:** ***This function performs bit-wise operations on numeric inputs.***

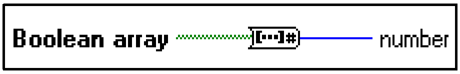
And Array Elements

Returns TRUE if all the elements in **Boolean array** are true; otherwise it returns FALSE.



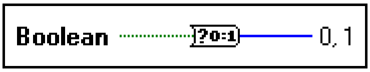
Boolean Array To Number

Converts **Boolean array** to an unsigned long integer by interpreting it as the two's complement representation of an integer with the 0th element of the array being the least significant bit.



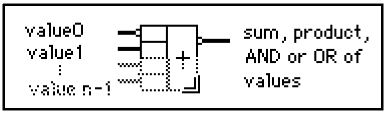
Boolean To (0,1)

Converts a Boolean value to a word integer--0 and 1 for the input values FALSE and TRUE, respectively.



Compound Arithmetic

Performs arithmetic on two or more numeric, cluster, or Boolean inputs.



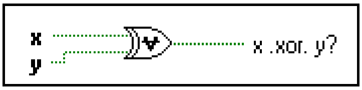
You select the operation (multiply, AND, or OR) by popping up on the function and selecting **Change Mode**.

You can invert the inputs or the output of this function by popping up on the individual terminals, and selecting **Invert**. For Add, select **Invert** to negate an input or the output. For Multiply, select **Invert** to use the reciprocal of an input or to produce the reciprocal of the output. For AND or OR, select **Invert** to logically negate an input or the output.

Note: ***You add inputs to this node by popping up on an input and selecting Add Input or by placing the Positioning tool in the lower left or right corner of the node and dragging it.***

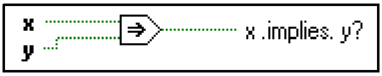
Exclusive Or

Computes the logical Exclusive OR of the inputs.



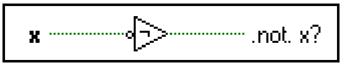
Implies

Computes the logical OR of **y** and of the logical negation of **x**. That is, the function negates **x** and then computes the logical OR of **y** and of the negated **x**.



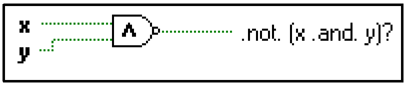
Not

Computes the logical negation of the input.



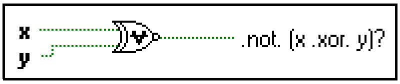
Not And

Computes the logical NAND of the inputs.



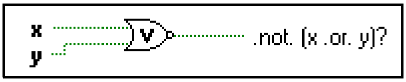
Not Exclusive Or

Computes the logical negation of the logical exclusive OR of the inputs.



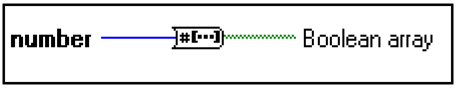
Not Or

Computes the logical NOR of the inputs.



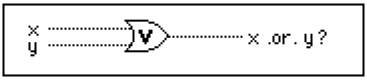
Number To Boolean Array

Converts **number** to a Boolean array of 8, 16, or 32 elements, where the 0th element corresponds to the least significant bit (LSB) of the two's complement representation of the integer.



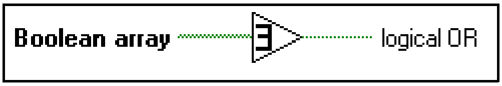
Or

Computes the logical OR of the inputs.



Or Array Elements

Returns FALSE if all the elements in **Boolean array** are false; otherwise it returns TRUE.



Boolean Constant

Use this to supply a constant true/false value to the block diagram. Set this value by clicking on the **T** or **F** portion of the constant with the Operating tool. This value cannot be changed while the VI executes.

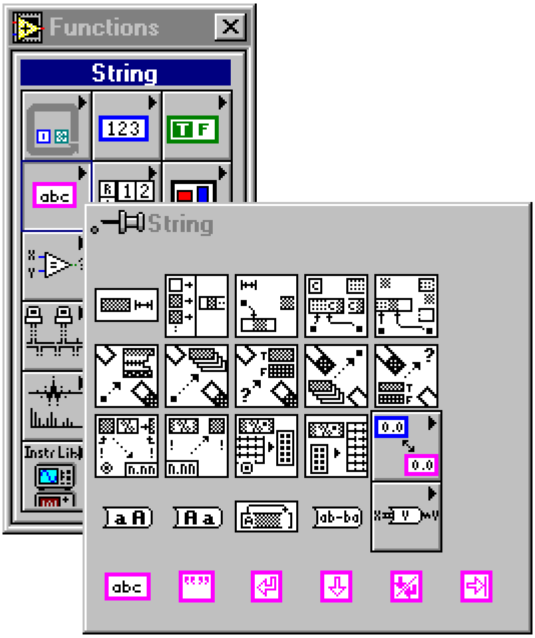
You can assign a label to this constant.

Chapter 6

String Functions

This chapter describes the string functions, including those that convert strings to numbers and numbers to strings.

The following illustration shows the **String** palette, which you access by selecting **Functions»String**.



Overview of Polymorphism for String Functions

This section provides descriptions of polymorphism for String functions, Additional String to Number functions, and String Conversion functions.

Polymorphism for String Functions

String Length, To Upper Case, To Lower Case, Reverse String, and Rotate String accept strings, clusters, arrays of strings, and arrays of clusters. To Upper Case and To Lower Case also accept numbers, clusters of numbers, and arrays of numbers, interpreting them as ASCII codes for characters (refer to the Appendix B, *Multiline Interface Messages*, later in this manual, for the numbers that correspond to each character). Width and precision inputs must be scalar.

Polymorphism for Additional String to Number Functions

To Decimal, To Hex, To Octal, To Engineering, To Fractional, and To Exponential accept clusters and arrays of numbers and produce clusters and arrays of strings. From Decimal, From Hex, From Octal, and From Exponential/Fract/Sci accept clusters and arrays of strings and produce clusters and arrays of numbers. Width and precision inputs must be scalar.

Polymorphism for String Conversion Functions

The Path To String and String To Path functions are polymorphic. That is, they work on scalar values, arrays of scalars, clusters of scalars, arrays of clusters of scalars, and so on. The output has the same composition as the input but with the new type.

Format Strings Overview

Many G functions accept a **format string** input, which controls the behavior of the function. A format string is composed of one or more format specifiers, which determine what action to take to process a given parameter. The Format Into String and Scan From String functions can use multiple format specifiers in the format string, one for each resizable input or output to the function. Characters in the string that are not part of the format specifier are copied verbatim to the output string (in the case of Format Into String) or are matched exactly in the input string (in the case of Scan From String), with the exception of special escape codes. You can use these codes to insert nondisplayable characters, the backslash, and percent characters within any format string. These codes are similar to those used in the C programming language.

Table 6-1 displays the special escape codes. A code does not exist for the platform-dependent end-of-line (eol) character. If you need to append one, use the End-of-Line constant from the **String** palette.

Table 6-1. Special Escape Codes

Code	Meaning
\r	Carriage Return
\t	Tab
\b	Backspace
\n	Newline
\f	Form Feed
\s	space
\xx	character with hexadecimal ASCII code xx (using 0 through 9 and upper case A through F)
\\\\	\\
%%	%

Notice also that for the Scan From String and Format & Strip functions, a space in the format string matches any amount of whitespace (spaces, tabs, and form feeds) in the input string.

The Format & Append, Format & Strip, Array To Spreadsheet String, and Spreadsheet String To Array functions use only one format specifier in the format string, because these functions have only one input that can be converted. Any extraneous specifiers inserted into these functions are treated as literal strings with no special meaning.

For functions that output a string, such as Format Into String, Format & Append, and Array To Spreadsheet String, a format specifier has the following syntax. Double brackets ([]) enclose optional elements.

```
%[-][+][^][0][Width][.Precision][{unit}]Conversion Code
```

For functions that scan a string, such as Scan From String, Format & Strip, and Spreadsheet String to Array, a format specifier has the following, simplified syntax:

```
%[Width]Conversion Code
```

Table 6-2 displays the string syntax available.

Table 6-2. String Syntax

Syntax Element	Description
%	Begins the formatting specification.
– (optional)	Causes the parameter to be left justified rather than right justified within its width.
+ (optional)	For numeric parameters, includes the sign even when the number is positive.
^ (optional	When used with the `e` or `g` conversion codes, uses engineering notation (exponent is always a multiple of 3).
`0` (optional)	Pads any excess space to the left of a numeric parameter with 0s rather than spaces.
Width (optional)	When scanning, specifies an exact field width to use. G scans only the specified number of characters when processing the parameter. When formatting, specifies the minimum character field width of the output. This is not a maximum width; G uses as many characters as necessary to format the parameter without truncating it. G pads the field to the left or right of the parameter with spaces, depending on justification. If Width is missing or zero, the output is only as long as necessary to contain the converted input parameter.
.	Separates Width from Precision.

Table 6-2. String Syntax (Continued)

Syntax Element	Description
Precision (optional)	For floating-point parameters, specifies the number of digits to the right of the decimal point. If Width is not followed by a period, G inserts a fractional part of six digits. If Width is followed by a period, and Precision is missing or 0, G does not insert a fractional part. For string parameters, specifies the maximum width of the field. G truncates strings longer than this length.
{unit} (optional)	Overrides the choice of unit of a VI when converting a physical quantity (a value with an associated unit). Must be a valid unit.
Conversion Codes	Single character that specifies how to convert **number**, as follows `d` to decimal integer `x` to hex integer `o` to octal integer `b` to binary integer `f` to floating-point number with fractional format `e` to floating-point number with scientific notation `g` to floating-point number using `e` format if the exponential is less than –4 or greater than Precision, or `f` format otherwise `s` to string

The Conversion Codes used in G are similar to those used in the C programming language. However, G uses conversion codes to determine the textual format of the parameter, not the datatype of the parameter.

You can use the `d`, `x`, `o`, `b`, `f`, `e` and `g` conversion codes to process any numeric G data type, including complex numbers and enums.

For complex numbers, you can use the format specifier to process both the real and imaginary parts as a single parameter.

You can use the `s` conversion code to process string or path parameters or enums.

Notice that you can use either a numeric or string conversion code with an enum, depending on whether you want the numeric value or symbolic (string) value of the enum.

For compatibility with C, G treats a `u` conversion code (unsigned integer) the same as a `d`, and ignores an `l` or `L` preceding the conversion code. However, in G it is the datatype of the parameter that determines the size of an integer and whether the integer is signed or unsigned.

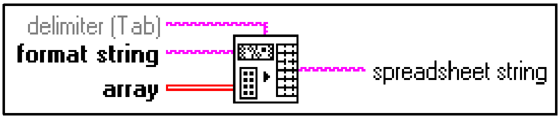
For examples of format string usage, see the Format Into String and Scan From String function descriptions later in this chapter.

String Function Descriptions

The following string functions are available.

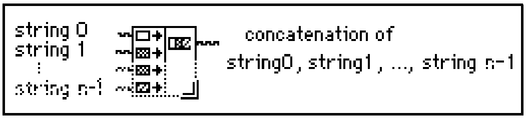
Array To Spreadsheet String

Converts an **array** of any dimension to **spreadsheet string**. **spreadsheet string** is a table in string form, containing delimiter-separated column elements, a platform-dependent EOL character separating rows, and, for arrays of three or more dimensions, pages are separated.



Concatenate Strings

Concatenates input strings and one-dimensional arrays of strings into a single, output string. For array inputs, this function concatenates each element of the array.



Format Into String

Converts input arguments into **resulting string**, whose format is determined by **format string**. You increase the number of parameters by popping up on the node and selecting **Add Parameter** or by placing the Positioning tool over the lower left or right corner of the node and then stretching it until you reach the desired number of arguments.

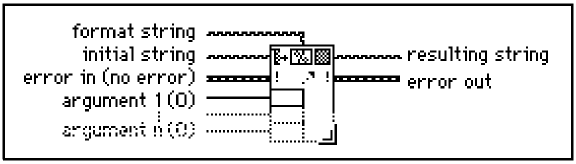


Table 6-3 shows the possible errors which may be produced in error out by Format Into String.

Table 6-3. Possible Format Into String Errors

Error	Code	Description
Format specifier type mismatch	81	The datatype of a format specifier in the format string does not match the datatype of the corresponding input argument.
Unknown format specifier	82	The format string contains an invalid format specifier.
Too few format specifiers	83	There are more arguments than format specifiers.
Too many format specifiers	84	There are more format specifiers than arguments.

☞ **Note:** ***If an error occurs, the source component of the error out cluster contains a string of the form "Format Into String (arg n)," where n is the first argument for which the error occurred.***

If you wire a block diagram constant string to format string, G checks for errors in format string at compile time. Such errors must be corrected before you can run the VI. In this case, no errors can occur at run time.

Format Specifier Examples

In Table 6-4, the underline character (_) represent spaces in the output. The last three entries are examples of physical quantity inputs.

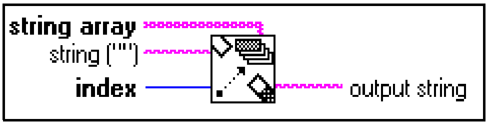
Table 6-4. Format Specifiers

Format String	Argument(s)	Resulting String
score= %2d%%	87	score= 87%
level= \n%–7.2e V	0.03642	level= 3.64e–2 V
Name: %s, %s.	Smith John	Name: Smith, John.
Temp: %05.1f %s	96.793 Fahrenheit	Temp: 096.8 Fahrenheit
String: %10.5s.	Hello, World	String:_____Hello.
%5.3f	5.67 N	5.670 N
%5.3{mN}f	5.67 N	5670.000 mN
%5.3{kg}f	5.67 N	5.670 ?kg

The last table entry shows the output when the unit in the format specifier is in conflict with the input unit.

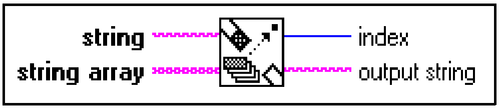
Index & Append

Selects a string specified by **index** from **string array** and appends that string to **string**.



Index & Strip

Compares each string in **string array** with the beginning of **string** until there is a match.



Match Pattern

Searches for **regular expression** in **string** beginning at **offset,** and if it finds a match, splits **string** into three substrings.

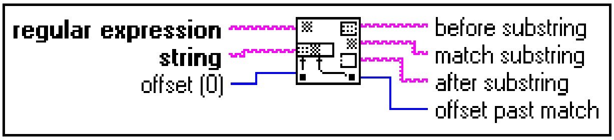


Table 6-5. Special Characters for Match Pattern

Special Character	Interpreted by the Match Pattern Function as...
.	Matches any character.
?	Matches zero or one instances of the expression preceding *?*.
\	Cancels the interpretation of special characters (for example, \? matches a question mark). You can also use the following constructions for the space and nondisplayable characters \b backspace \f form feed \n newline \s space \r carriage return \xx any character, where *xx* is the hex code using 0 through 9 and upper case A through F \t tab
^	If ^ is the first character of **regular expression**, it anchors the match to the **offset** in **string**. The match fails unless **regular expression** matches that portion of **string** that begins with the character at **offset**. If ^ is not the first character, it is treated as a regular character.

Table 6-5. Special Characters for Match Pattern (Continued)

Special Character	Interpreted by the Match Pattern Function as...
[ ]	Encloses alternates. For example, [abc] matches a, b, or c. The following character has special significance when used within the brackets in the following manner. – (dash)Indicates a range when used between digits, or lowercase or uppercase letters (for example, [0–5],[a–g], or [L–Q]) The following characters have significance only when they are the first character within the brackets. ~ Excludes the set of characters, including nondisplayable characters. [~0–9] matches any character other than 0 through 9. ^ Excludes the set with respect to all the displayable characters (and the space characters). [^0–9] gives the space characters and all displayable characters except 0 through 9.
+	Matches the longest number of instances of the expression preceding +; there must be at least one instance to constitute a match.
*	Matches the longest number of instances of the expression preceding * in **regular expression**, including zero instances.
$	If $ is the last character of **regular expression**, it anchors the match to the last element of **string**. The match fails unless **regular expression** matches up to and including the last character in the string. If $ is not last, it is treated as a regular character.

Table 6-6 shows examples of the Strings for the Match Pattern functions.

Table 6-6. Strings for the Match Pattern Examples

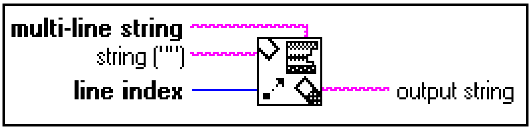
Characters to Be Matched	Regular Expression
VOLTS	VOLTS
All uppercase and lowercase versions of volts, that is, VOLTS, Volts, volts, and so on	[Vv][Oo][Ll][Tt][Ss]

Table 6-6. Strings for the Match Pattern Examples (Continued)

Characters to Be Matched	Regular Expression
A space, a plus sign, or a minus sign	[+–]
A sequence of one or more digits	[0–9]+
Zero or more Spaces	\s * or * (that is, a space followed by an asterisk)
One or more Spaces, Tabs, Newlines, or Carriage Returns	[\t \r \n \s]+
One or more characters other than digits	[~0–9]+
The word Level only if it begins at the offset position in the string	^Level
The word Volts only if it appears at the end of the string	Volts$
The longest string within parentheses	(.*)
The longest string within parentheses but not containing any parentheses within it	([~( )]*)
The character, [	[[ ]

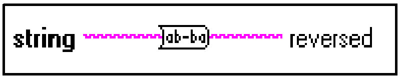
Pick Line & Append

Chooses a line from **multi-line string** and appends that line to **string**.



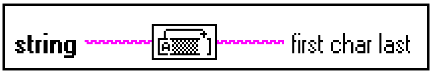
Reverse String

Produces a string whose characters are in reverse order of those in **string**.



Rotate String

Places the first character of **string** in the last position of **first char last**, shifting the other characters forward one position. For example, the string *abcd* becomes *bcda*.



Scan From String

Scans the input string and converts the string according to **format string**. You increase the number of parameters by popping up on the node and selecting **Add Parameter** or by placing the Positioning tool over the lower left or right corner of the node and then stretching it until you reach the desired number of parameters.

Use Scan From String when you know the exact format of the input string.

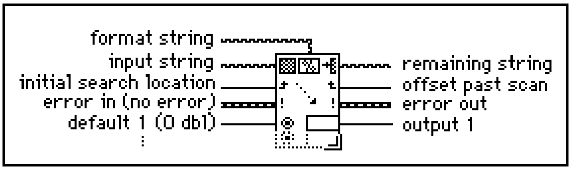


Table 6-7 lists the Scan from String errors.

Table 6-7. Scan From String Errors

Error	Code	Description
Format specifier type mismatch	81	The datatype of a format specifier in the format string does not match the datatype of the corresponding output.
Unknown format specifier	82	The format string contains an invalid format specifier.
Too few format specifiers	83	There are more arguments than format specifiers.

Table 6-7. Scan From String Errors (Continued)

Error	Code	Description
Too many format specifiers	84	There are more format specifiers than arguments.
Scan failed	85	Scan From String was unable to convert the input string into the datatype indicated by the format specifier.

☞ **Note:** ***If an error occurs, the source component of the*** **error out** ***cluster contains a string of the form "Scan From String (arg n)," where n is the first argument for which the error occurred.***

If you wire a block diagram constant string to format string, G checks for errors in format string at compile time. You must correct these errors before you can run the VI. In this case, only Scan failed can occur at run time.

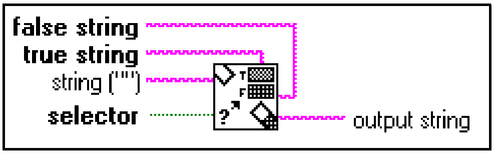
Table 6-8 lists Scan From String examples.

Table 6-8. Scan from String Examples

Input String	Format String	Default(s)	Output(s)	Remaining String
abc xyz 12.3+56i 7200	%s %s%f%2d		abc xyz 12.3+56i 72	00
Q+1.27E–3 tail	Q%f t		1.27E–3	ail
0123456789	%3d%3d		12 345	6789
X:9.860 Z:3.450	X:%fY:%f	100 (I32) 100.0 (DBL)	10 100.0	Z: 3450
set49.4.2	set%d		49	.4.2

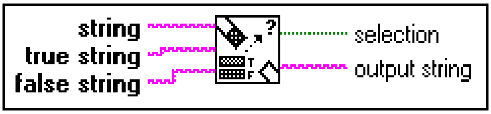
Select & Append

Selects either a **false string** or **true string** according to a Boolean **selector** and appends that string to **string**.



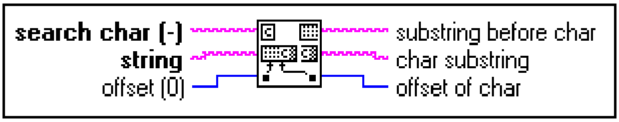
Select & Strip

Examines the beginning of **string** to see whether it matches **true string** or **false string**. This function returns a Boolean TRUE or FALSE value in **selection**, depending on whether **string** matches **true string** or **false string**.



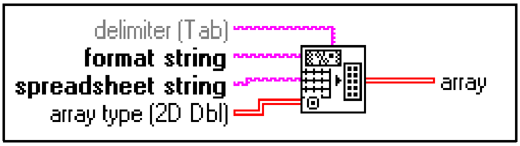
Split String

Splits the string at offset or searches for the first occurrence of **search char** in the **string**, beginning at **offset**, and splits the string at that point.



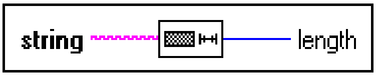
Spreadsheet String To Array

Converts the **spreadsheet string** to a numeric **array** of the dimension and representation of **array type**. This function works for arrays of strings as well as arrays of numbers.



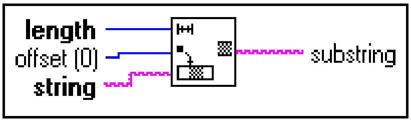
String Length

Returns in **length** the number of characters (bytes) in **string**.



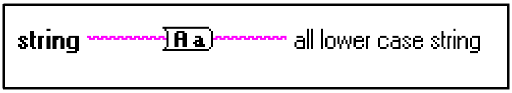
String Subset

Returns the **substring** of the original **string** beginning at **offset** and containing **length** number of characters.



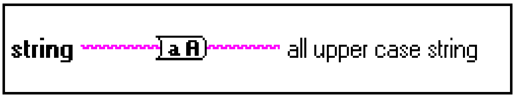
To Lower Case

Converts all alphabetic characters in **string** to lowercase characters. This function does not affect nonalphabetic characters.



To Upper Case

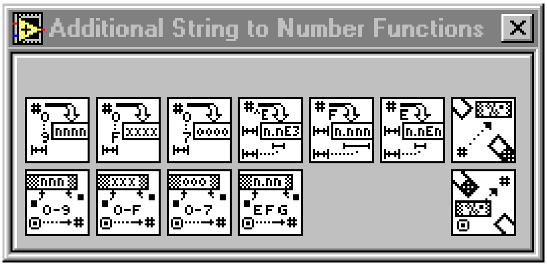
Converts all alphabetic characters in **string** to uppercase characters. This function does not affect nonalphabetic characters.



Additional String To Number Function Descriptions

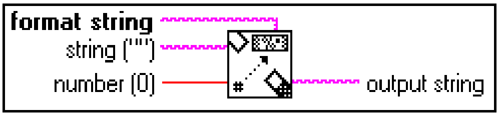
For general information about Additional String to Number functions, see *Polymorphism for Additional String to Number Functions*, earlier in this chapter.

The following illustration displays the options available on the **Additional String to Number Functions** subpalette.



Format & Append

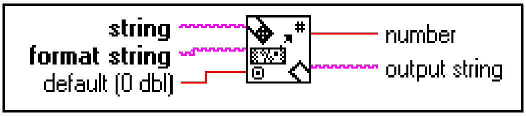
Converts **number** into a regular string according to the format specified in **format string,** and appends this to **string**.



☞ **Note:** ***The Format Into String function has the same functionality as Format & Append but can use multiple inputs, so that you can convert information simultaneously. You should consider using Format Into String instead of this function: in many cases, this can simplify your block diagram.***

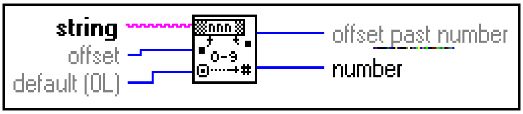
Format & Strip

Looks for **format string** at the beginning of **string**, formats any number in this string portion according to the conversion codes in **format string**, and returns the converted number in **number** and the remainder of **string** after the match in **output string**.



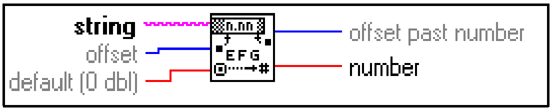
From Decimal

Converts the numeric characters in **string**, starting at **offset**, to a decimal integer and returns it in **number**.



From Exponential/Fract/Eng

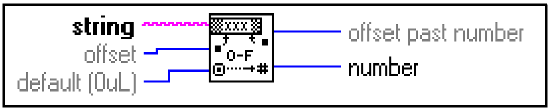
Interprets the characters 0 through 9, plus, minus, e, E, and the decimal point (usually period) in **string** starting at **offset** as a floating-point number in engineering notation, or exponential or fractional format and returns it in **number**.



☞ **Note:** ***If you wire the characters Inf or NaN to string, this function returns the G values Inf and NaN, respectively.***

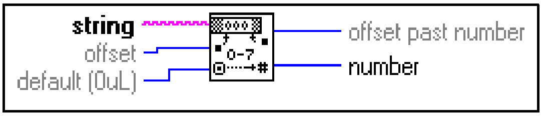
From Hexadecimal

Interprets the characters 0 through 9, A through F, and a through f in **string** starting at **offset** as a hex integer and returns it in **number**.



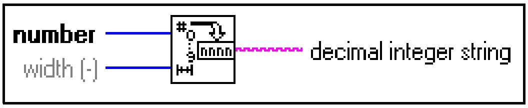
From Octal

Interprets the characters 0 through 7 in **string** starting at **offset** as an octal integer and returns it in **number**. This function also returns the index in **string** of the first character following the number.



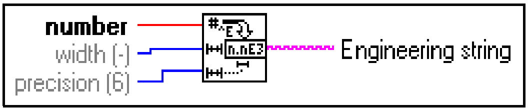
To Decimal

Converts **number** to a string of decimal digits **width** characters wide, or wider if necessary.



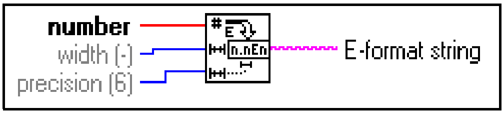
To Engineering

Converts **number** to an engineering format, floating-point string **width** characters wide, or wider if necessary. Engineering format is similar to E format, except the exponent is a multiple of three (–3, 0, 3, 6).



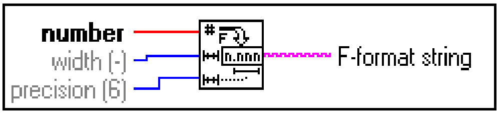
To Exponential

Converts **number** to an E-format (exponential notation), floating-point string **width** characters wide, or wider if necessary.



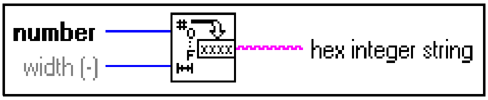
To Fractional

Converts **number** to an F-format (fractional notation), floating-point string **width** characters wide, or wider if necessary.



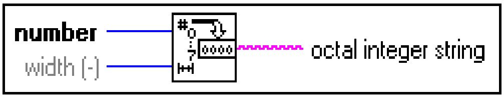
To Hexadecimal

Converts **number** to a string of hexadecimal digits **width** characters wide, or wider if necessary.



To Octal

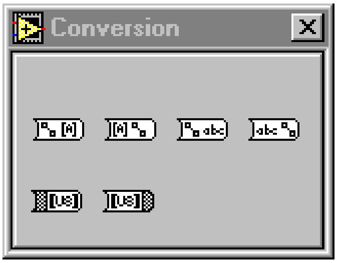
Converts **number** to a string of octal digits **width** characters wide, or wider if necessary.



String Conversion Function Descriptions

For general information about String Conversion functions, see *Overview of Polymorphism for String Functions* earlier in this chapter.

The following illustration shows the **String Conversion** subpalette.

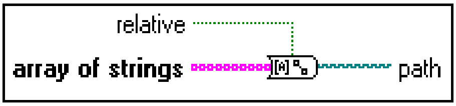


Array Of Strings To Path accepts one-dimensional (1D) arrays of strings, Path To Array Of Strings accepts paths, Path To String accepts paths, and String To Path accepts strings.

Array Of Strings To Path

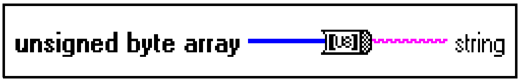
Converts an **array of strings** into a relative or absolute **path**.

If you have an empty string in the array the directory location before the empty string is deleted in the path output. Think of this as moving up a level in directory hierarchy.



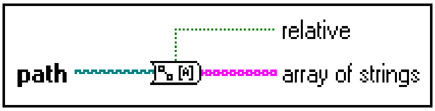
Byte Array To String

Converts an array of unsigned bytes into a **string**.



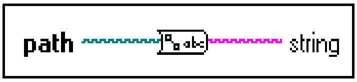
Path To Array Of Strings

Converts a **path** into an **array of strings** and indicates whether the path is **relative**.



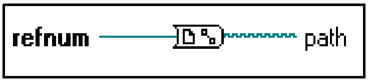
Path To String

Converts **path** into a string describing a path in the standard format of the platform.



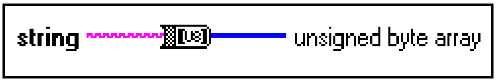
Refnum To Path

Returns the **path** associated with the specified **refnum.**



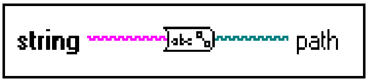
String To Byte Array

Converts a **string** into an array of unsigned bytes.



String To Path

Converts a string, describing a path in the standard format for the current platform, to a path.



String Fixed Constants

The following String Fixed Constants are available.

String Constant

Use this to supply a constant ASCII value to the block diagram. Set this value by clicking inside the constant with the Operating tool and typing in the value. You can change the display mode so you can see non-displayable characters or the hex equivalent to the characters. You can also set the constant in password display mode so "*" are displayed when you type in characters.

The value of the string constant cannot be changed while the VI executes. You can assign a label to this constant.

Carriage Return

Consists of a constant string containing the ASCII CR value.

Empty String

Consists of a constant string that is empty. Length is zero.

End of Line

Consists of a constant string containing the platform-dependent, end of line value. For Windows, the value is CRLF; for Macintosh, the value is CR; and on UNIX, the value is LF.

Line Feed

Consists of a constant string containing the ASCII LF value.

Tab

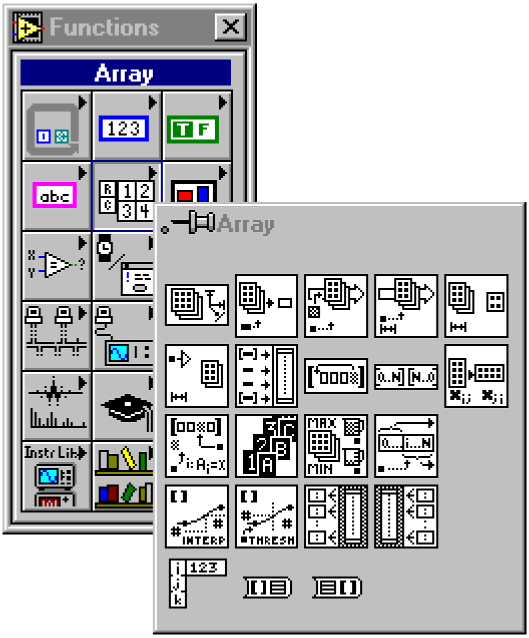
Consists of a constant string containing the ASCII HT (horizontal tab) value.

Array Functions

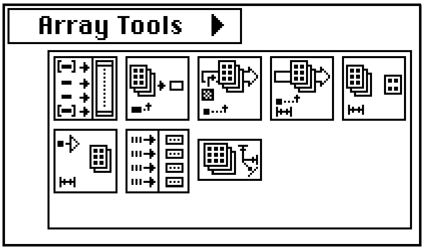
Chapter 7

This topic describes the functions for array operations.

The following illustration shows the **Array** palette which you access by selecting **Functions»Array**.



Some of the array functions are also available from the **Array Tools** palette of most terminal or wire pop-up menus. The illustration below shows the pop-up menu.



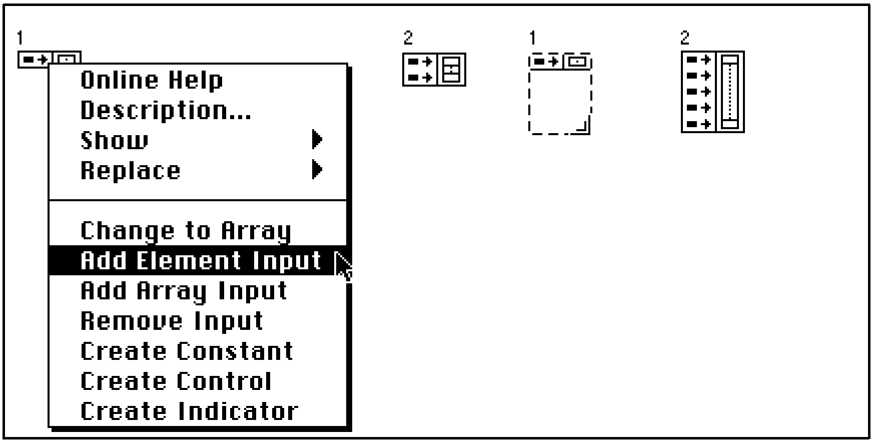
If you select the functions from this palette, they appear with the correct number of terminals to wire to the object on which you popped up.

For examples of array functions, see `examples\general\arrays.llb`.

Array Function Overview

Some of the array functions have a variable number of terminals. When you drop a new function of this kind, it appears on the block diagram with only one or two terminals. You can add and remove terminals by using the pop-up menu **Add Element Input** or **Add Array Input** and **Remove Input** commands (the actual names depend on the function) or by resizing the node vertically from any corner. If you want to add terminals by popping up, you must place your cursor on the input terminals to access the pop-up menu.

You can shrink the node if doing so does not delete wired terminals. The **Add Element Input** or **Add Array Input** command inserts a terminal directly after the one on which you popped up. The **Remove Input** command removes the terminal on which you popped up, even if it is wired. The following illustration shows the two ways to add more terminals to the Build Array function.



Out-of-Range Index Values

Attempting to index an array beyond its bounds results in a default value determined by the array element type.

Polymorphism for Array Functions

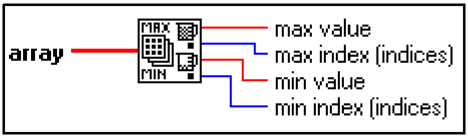
Most of the array functions accept n-dimensional arrays of any type, however the wiring diagrams in the function descriptions show numeric arrays as the default data type.

Array Function Descriptions

The following Array functions are available.

Array Max & Min

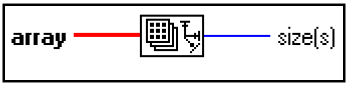
Searches for the first maximum and minimum values in **numeric array**. This function also returns the indices where it finds the maximum and minimum values.



The function compares each datatype according to the rules referred to in Chapter 9, *Comparison Functions*.

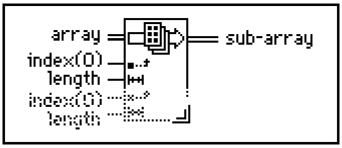
Array Size

Returns the number of elements in each dimension of **array**.



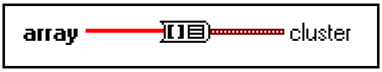
Array Subset

Returns a portion of **array** starting at **index** and containing **length** elements.



Array To Cluster

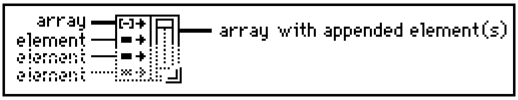
Converts a 1D array to a cluster of elements of the same type as the array elements. Pop up on the node to set the number of elements in the cluster. The default is nine. The maximum cluster size for this function is 256.



For more information on clusters, see Chapter 8, *Cluster Functions*.

Build Array

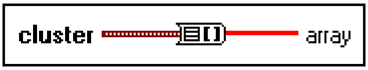
Appends any number of array or element inputs in top-to-bottom order to create **array with appended element**.



To change an element input to an array input, pop up on the input and select **Change to Array**. In general, to build an array of *n*-dimensions, each **array** input must be of the same dimension, *n*, and each **element** input must have *n*–1 dimensions. To create a 1D array, connect scalar values to the element inputs and 1D arrays to the array inputs. To build a 2D array, connect 1D arrays to element inputs and 2D arrays to the array inputs.

Cluster To Array

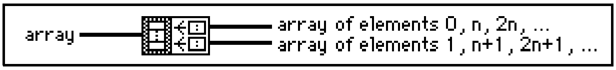
Converts a cluster of identically typed components to a 1D array of elements of the same type.



For more information on clusters, see Chapter 8, *Cluster Functions*.

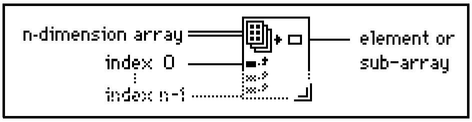
Decimate 1D Array

Divides the elements of **array** into the output arrays.


array
array of elements 0, n, 2n, ...
array of elements 1, n+1, 2n+1, ...

Index Array

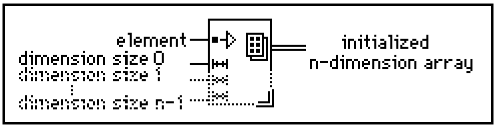
Returns the **element** of **array** at **index**. If **array** is multidimensional, you must add additional **index** terminals for each dimension of the **array**.



In addition to extracting an element of the array, you can *slice* out a higher dimensional component by disabling one or more of the index terminals.

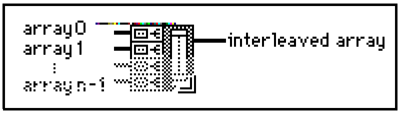
Initialize Array

Creates an *n*-dimensional array in which every element is initialized to the value of **element**.



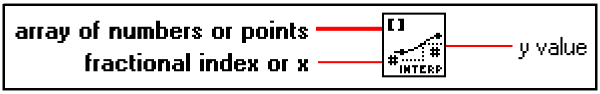
Interleave 1D Arrays

Interleaves corresponding elements from the input arrays into a single output array.



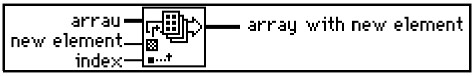
Interpolate 1D Array

Uses the integer part of the **fractional index of x** to index the array and the fractional part of **fractional index of x** to linearly interpolate between the values of the indexed element and its adjacent element.



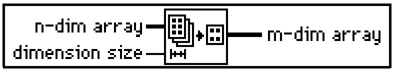
Replace Array Element

Replaces the element in **array** at **index** with the **new element**.



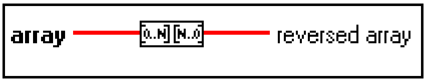
Reshape Array

Changes the dimension of an array according to the value of **dimension size**. For example, you can use this function to change a 1D array into a 2D array or vice versa. You can also use it to increase and decrease the size of a 1D array.



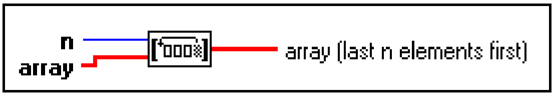
Reverse 1D Array

Reverses the order of the elements in **array**.



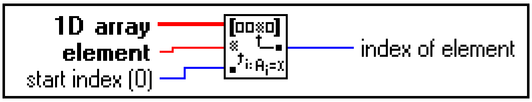
Rotate 1D Array

Rotates the elements of **array** by the number of places and in the direction indicated by **n**.



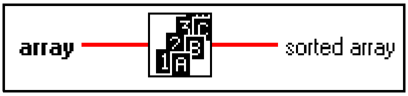
Search 1D Array

Searches for **element** in **1D array** starting at **start index**.



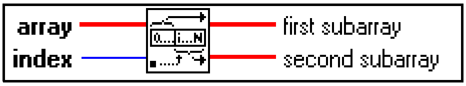
Sort 1D Array

Returns a sorted version of **array** with the elements arranged in ascending order. The rules for comparing each datatype are described in Chapter 9, *Comparison Functions*.



Split 1D Array

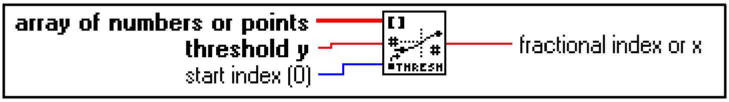
Divides **array** at **index** and returns the two portions.



Threshold 1D Array

Compares **threshold y** to the values in **array of numbers or points** starting at **start index** until it finds a pair of consecutive elements such that **threshold y** is greater than the value of the first element and less than or equal to the value of the second element.

The function then calculates the fractional distance between the first value and **threshold y** and returns the fractional index at which **threshold y** would be placed within **array of numbers or points** using linear interpolation.

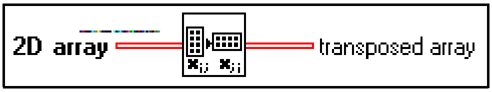


For example, suppose **array of numbers or points** is an array of four numbers [4, 5, 5, 6], **start index** is 0, and **threshold y** is 5. The **fractional index or x** is 1, corresponding to the index of the first value of 5 the function finds. Suppose the array elements are 6, 5, 5, 7, 6, 6, the **start index** is 0, and the **threshold y** is 6 or less. The output is 0. If **threshold y** is greater than 7 for the same set of numbers, the output is 5. If **threshold y** is 14.2, **start index** is 5, and the values in the array starting at index 5 are 9.1, 10.3, 12.9, and 15.5, **threshold y** falls between elements 7 and 8 because 14.2 is midway between 12.9 and 15.5. The value for **fractional index or x** is 7.5, that is, halfway between 7 and 8.

If the array input consists of an array of points where each point is a cluster of x and y coordinates, the output is the interpolated x value corresponding to the interpolated position of **threshold y** rather than the fractional index of the array. If the interpolated position of **threshold y** is midway between indices 4 and 5 of the array with x values of –2.5 and 0 respectively, the output is not an index value of 4.5 as it would be for a numeric array, but rather an x value of –1.25.

Transpose 2D Array

Rearranges the elements of **2D array** such that **2D array**[*i*,*j*] becomes **transposed array**[*j*,*i*].

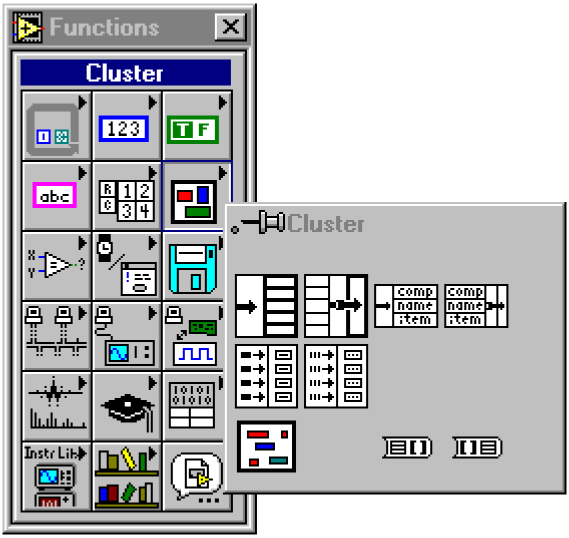


Cluster Functions

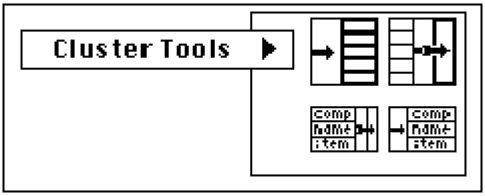
Chapter 8

This chapter describes the functions for cluster operations.

The following illustration shows the **Cluster** palette, which you access by selecting **Functions»Cluster**.



Some of the cluster functions are also available from the **Cluster Tools** palette of most terminal or wire pop-up menus. The following illustration shows the pop-up menu.



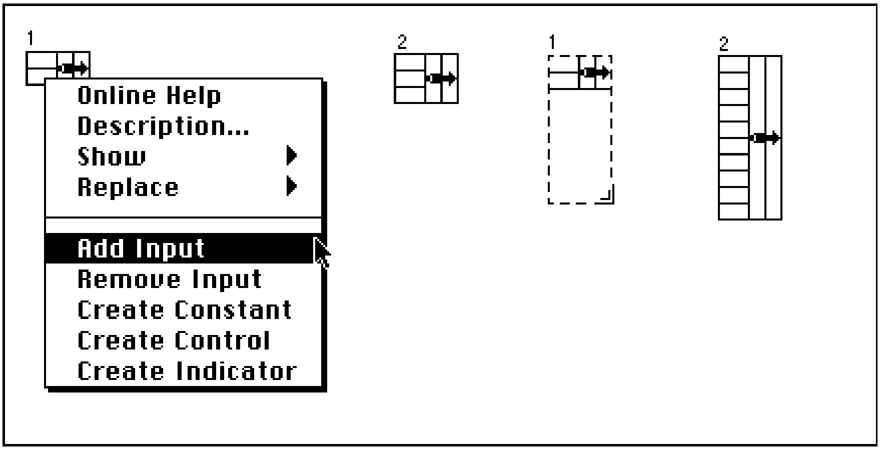
If you select the functions from this palette, they appear with the correct number of terminals to wire to the object on which you popped up.

Cluster Function Overview

Some of the cluster functions have a variable number of terminals. When you drop a new function of this kind, it appears on the block diagram with only one or two terminals. You can add and remove terminals by using the pop-up menu **Add Input** or **Remove Input** options or by resizing the node using the Positioning tool. If you want to add terminals by popping up, you must place your cursor on the input terminal to access the pop-up menu.

You can shrink the node if doing so does not delete wired terminals. The **Add Input** option inserts a terminal directly after the one on which you popped up. The **Remove Input** option removes the terminal on which you popped up, even if it is wired.

The following illustration shows the two ways to add more terminals to the Bundle function.



Polymorphism for Cluster Functions

The Bundle and Unbundle functions do not show the datatype for their individual input or output terminals until you wire objects to these terminals. When you wire them, these terminals look similar to the datatypes of the corresponding front panel control or indicator terminals.

Setting the Order of Cluster Elements

Cluster elements have a logical order that is unrelated to their positions within the shell. The first object you insert in the cluster is element 0, the second is 1, and so on. If you delete an element, the order adjusts automatically. You can change the current order by selecting the **Cluster Order...** option from the cluster pop-up menu.

Clicking on an element with the cluster order cursor sets the elements place in the cluster order to the number displayed inside the Tools palette. You change this order by typing a new number into that field. When the order is as you want it, click on the **Enter** button to set it and exit the cluster order edit mode. Click on the **X** button to revert to the old order.

The cluster order determines the order in which the elements appear as terminals on the Bundle and Unbundle functions in the block diagram.

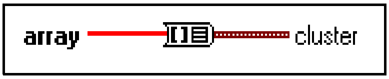
The Bundle By Name and Unbundle By Name functions give you more flexible access to data in clusters. With these functions, you can access specific elements in clusters by name and access only the elements you want to access. Because these functions reference components by name and not by cluster position, you can change the data structure of a cluster without breaking wires, as long as you do not change the name of or remove the component you reference on the block diagram.

Cluster Function Descriptions

The following cluster functions are available.

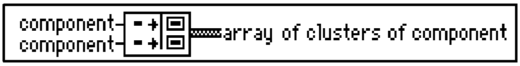
Array To Cluster

Converts a 1D array to a cluster of elements of the same type as the array elements. Pop up on the node or resize it to set the number of elements in the cluster. The default is nine. The maximum cluster size for this function is 256.



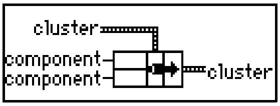
Build Cluster Array

Assembles all the **component** inputs in top-down order into an array of clusters of that **component**. If the input is four, single-precision, floating-point components, the output is a four-element array of clusters containing one single-precision, floating-point number. Element 0 of the array has the value of the top component, and so on.



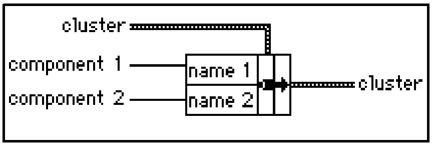
Bundle

Assembles all the individual input components into a single cluster.



Bundle By Name

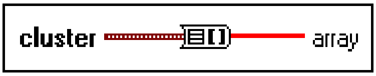
Replaces components in an existing cluster. After you wire the node to a cluster, you pop-up on the name terminals to choose from the list of components of the cluster.



You must always wire the **cluster** input. If you are creating a cluster for a cluster indicator, you can wire a local variable of that indicator to the **cluster** input. If you are creating a cluster for a cluster control of a subVI, you can place a copy of that control (possibly hidden) on the front panel of the VI and wire the control to the **cluster** input.

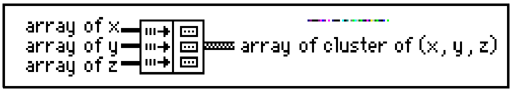
Cluster To Array

Converts a cluster of identically typed components to a 1D array of elements of the same type.

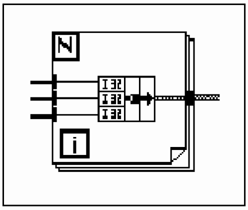


Index & Bundle Cluster Array

Indexes a set of arrays and creates a cluster array in which the *i*th element contains the *i*th element of each input array.

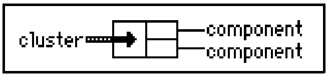


This function is equivalent to the following block diagram and is useful for converting a cluster of arrays to an array of clusters.



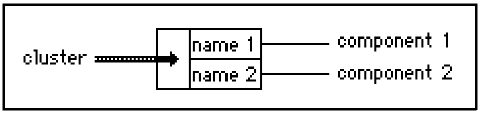
Unbundle

Disassembles a cluster into its individual components.



Unbundle By Name

Returns the cluster elements whose names you specify. You select the element you want to access by popping up on the name output terminals and selecting a name from the list of elements in the cluster.

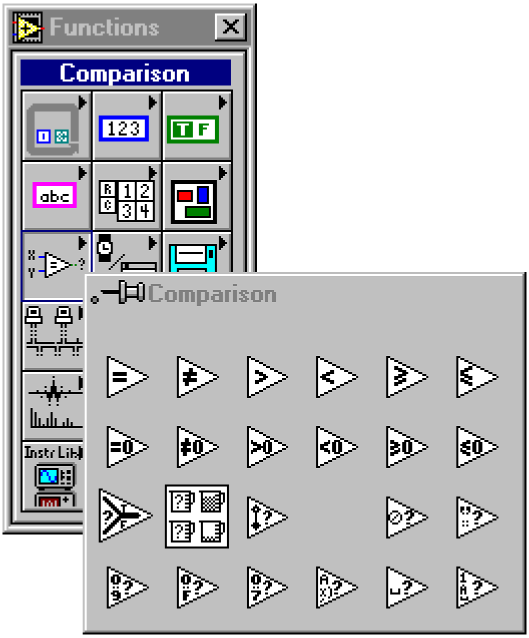


Chapter 9

Comparison Functions

This chapter describes the functions that perform comparisons or conditional tests.

The following illustration shows the **Comparison** palette, which you access by selecting **Functions»Comparison**.



For examples of comparison functions, see `examples\general\struct.llb`.

Comparison Function Overview

This section introduces the Comparison functions.

Compare Boolean

For the Compare Boolean functions, the Boolean value TRUE is greater than the Boolean value FALSE.

Compare Strings

These functions compare strings according to the numerical equivalent of the ASCII characters. Thus, `a` (with a decimal value of 97) is greater than `A` (65), which is greater than the numeral `0` (`48`), which is greater than the space character (`32`). These functions compare characters one by one from the beginning of the string until an inequality occurs, at which time the comparison ends. For example, LabVIEW evaluates the strings `abcd` and `abef` until it finds `c`, which is greater than the value of `e`. The presence of a character is greater than the absence of one. Thus, the string `abcd` is greater than `abc` because the first string is longer. Most of the comparison functions test one input or compare two inputs and return a Boolean value. The functions convert numbers to the same representation before comparing them. Comparisons with a value of NaN (not a number) return a value that indicates inequality.

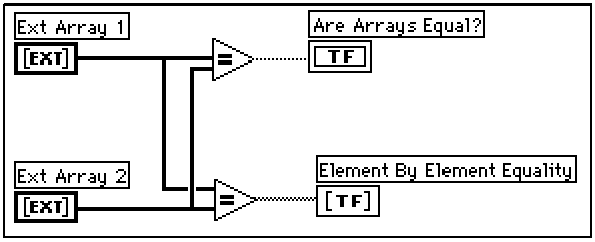
The functions that test the category of a string character (for example, the Decimal Digit? and Printable? functions) evaluate only the first character of the string.

Compare Clusters

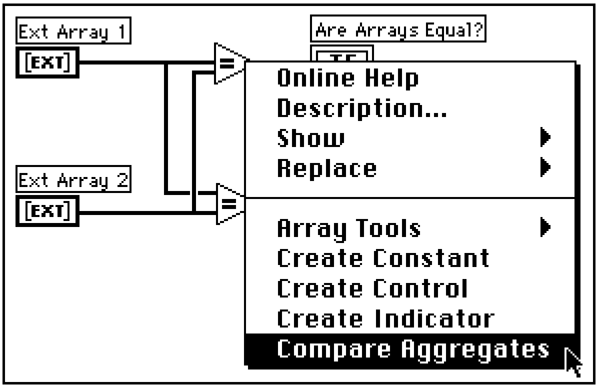
The comparison functions compare clusters the same way they compare strings, one element at a time starting with the 0th element until an inequality occurs. Clusters must have the same number of elements, of the same type, and in the same order if you want to compare them.

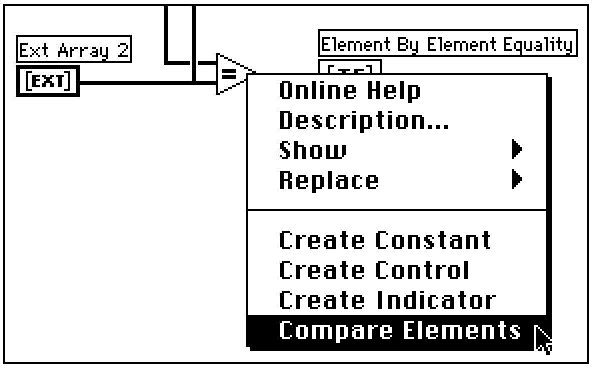
Compare Modes

Some of the comparison functions have two modes for comparing arrays or clusters. In the **Compare Aggregates** mode, if you compare two arrays or clusters, the function returns a single value. In the **Compare Elements** mode, the function compares the elements individually and then returns an array or cluster of Boolean values. The following illustration shows the two modes.



You change the comparison mode by selecting **Compare Elements** or **Compare Aggregates** in the pop-up menu for the node, as shown in the following illustrations.





When you compare two arrays of unequal lengths in the **Compare Elements** mode, LabVIEW ignores each element in the larger array whose index is greater than the index of the last element in the smaller array.

When you use the **Compare Aggregates** mode to compare two arrays, the following occurs: (1) LabVIEW searches for the first set of corresponding elements in the two inputs that differ, and uses those to determine the results of the comparison. (2) If all elements are identical except that one has more elements, LabVIEW considers the longer array to be greater than the shorter array. (3) If no elements of the two arrays differ, and the arrays have the same length, the arrays are equal. Thus, LabVIEW considers the array [1,2,3] to be greater than the array [1,2] and returns a single Boolean value in the **Compare Aggregates** mode.

When comparing clusters using the **Compare Aggregates** mode, LabVIEW goes by cluster order instead of array order. The two clusters LabVIEW compares are always the same length.

In the **Compare Elements** mode, LabVIEW returns a Boolean for each of the first two elements and ignores the last element of the larger array, as in the preceding example.

Arrays must have the same dimension size (for example, both two-dimensional), and for the comparison between multidimensional arrays to make sense, each dimension must have the same size.

The comparison functions that do not have the **Compare Aggregates** or **Compare Elements** modes compare arrays in the same manner as strings—one element at a time starting with the 0th element until an inequality occurs.

Character Comparison

You can use the functions that compare characters to determine a character’s type. The following functions are character comparison functions.

- Decimal Digit?
- Hex Digit?
- Lexical Class
- Octal Digit?
- Printable?
- White Space?

If the input is a string, the functions test the first character. If the input is an empty string, the result is FALSE. If the input is a number, the functions interpret it as a code for an ASCII character.

See Appendix B, *Multiline Interface Messages*, for the numbers that correspond to each character.

Polymorphism for Comparison Functions

The functions Equal?, Not Equal?, and Select take inputs of any type, as long as the inputs are the same type.

The functions Greater or Equal?, Less or Equal?, Less?, Greater?, Max & Min, and In Range? take inputs of any type except complex, path, or refnum, as long as the inputs are the same type. You can compare numbers, strings, Booleans, arrays of strings, clusters of numbers, clusters of strings, and so on. You cannot, however, compare a number to a string or a string to a Boolean, and so on.

The functions that compare values to zero accept numeric scalars, clusters, and arrays of numbers. These functions output Boolean values in the same data structure as the input.

The Not A Number/Path/Refnum function accepts the same input types as functions that compare values to zero. This function also accepts paths and refnums. Not A Number/Path/Refnum outputs Boolean values in corresponding structures. See Chapter 30, *Introduction to LabVIEW Instrument Driver VIs*, and Chapter 11, *File Functions*, for more information on these functions.

The functions Decimal Digit?, Hex Digit?, Octal Digit?, Printable?, and White Space? accept a scalar string or number input, clusters of strings or non-complex numbers, arrays of strings or non-complex numbers, and so on. The output consists of Boolean values in the same data structure as the input.

The function Empty String/Path? accepts a path, a scalar string, clusters of strings, arrays of strings, and so on. The output consists of Boolean values in the same data structure as the input.

You can use the Equal?, Not Equal?, Not A Number/Path/Refnum?, Empty String/Path?, and Select functions with paths and refnums, but no other comparison functions accept paths or refnums as inputs.

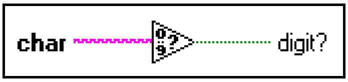
Comparison functions that use arrays and clusters normally produce Boolean arrays and clusters of the same structure. You can pop-up and change to compare aggregates, in which case the function outputs a single Boolean value. The function compares aggregates by comparing the first set of elements to produce the output, unless the first elements are equal, in which case the function compares the second set of elements, and so on.

Comparison Function Descriptions

The following Comparison functions are available.

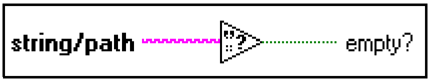
Decimal Digit?

Returns TRUE if **char** is a decimal digit ranging from 0 through 9. Otherwise, this function returns FALSE.



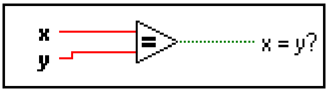
Empty String/Path?

Returns TRUE if **string/path** is an empty string or path. Otherwise, this function returns FALSE.



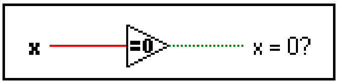
Equal?

Returns TRUE if **x** is equal to **y**. Otherwise, this function returns FALSE.



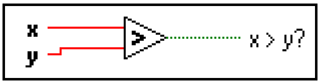
Equal To 0?

Returns TRUE if **x** is equal to 0. Otherwise, this function returns FALSE.



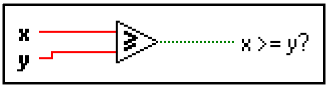
Greater?

Returns TRUE if **x** is greater than **y**. Otherwise, this function returns FALSE.



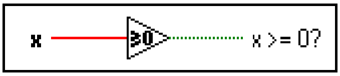
Greater Or Equal?

Returns TRUE if **x** is greater than or equal to **y**. Otherwise, this function returns FALSE.



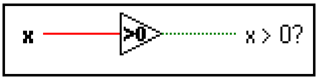
Greater Or Equal To 0?

Returns TRUE if **x** is greater than or equal to 0. Otherwise, this function returns FALSE.



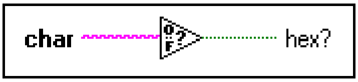
Greater Than 0?

Returns TRUE if **x** is greater than 0. Otherwise, this function returns FALSE.



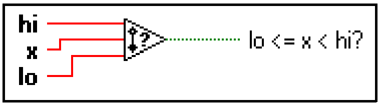
Hex Digit?

Returns TRUE if **char** is a hex digit ranging from 0 through 9, A through F, or a through f. Otherwise, this function returns FALSE.



n Range?

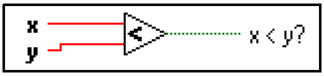
Returns TRUE if **x** is greater than or equal to **lo** and less than **hi**. Otherwise, this function returns FALSE.



☞ **Note:** ***This function always operates in the*** **Compare Aggregates** ***mode. To produce a Boolean array as an output, you must execute this function in a loop structure.***

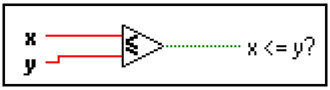
Less?

Returns TRUE if **x** is less than **y**. Otherwise, this function returns FALSE.



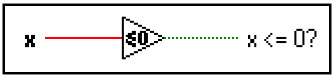
Less Or Equal?

Returns TRUE if **x** is less than or equal to **y**. Otherwise, this function returns FALSE.



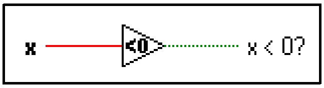
Less Or Equal To 0?

Returns TRUE if **x** is less than or equal to 0. Otherwise, this function returns FALSE.



Less Than 0?

Returns TRUE if **x** is less than 0. Otherwise, this function returns FALSE.



Lexical Class

Returns the **class number** for **char**.

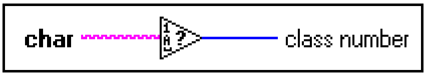
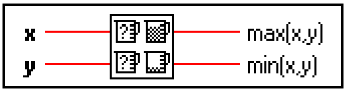


Table 9-1. Lexical Class Number Descriptions

Class Number	Lexical Class
0	Extended characters with a Command- or Option- key prefix (codes 128 through 255)
1	Nondisplayable ASCII characters (codes 0 to 31 excluding 9 through 13)
2	White space characters: Space, Tab, Carriage Return, Form Feed, Newline, and Vertical Tab (codes 32, 9, 13, 12, 10, and 11, respectively)
3	Digits 0 through 9
4	Uppercase characters A through Z
5	Lowercase characters a through z
6	All printable ASCII nonalphanumeric characters

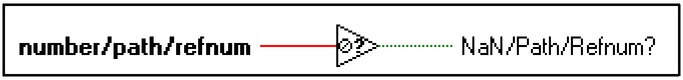
Max & Min

Compares **x** and **y** and returns the larger value at the top output terminal and the smaller value at the bottom output terminal.



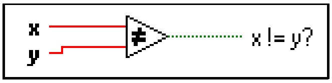
Not A Number/Path/Refnum?

Returns TRUE if **number/path/refnum** is not a numeric value, path, or refnum. Otherwise, this function returns FALSE. NaN can be the result of dividing by 0, the square root of a negative number, and so on.



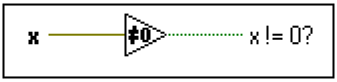
Not Equal?

Returns TRUE if **x** is not equal to **y**. Otherwise, this function returns FALSE.



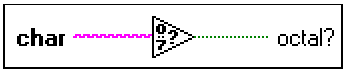
Not Equal To 0?

Returns TRUE if **x** is not equal to 0. Otherwise, this function returns FALSE.



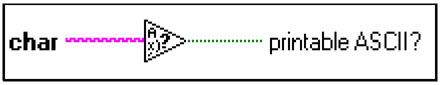
Octal Digit?

Returns TRUE if **char** is an octal digit ranging from 0 through 7. Otherwise, this function returns FALSE.



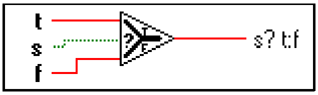
Printable?

Returns TRUE if **char** is a printable ASCII character. Otherwise, this function returns FALSE.



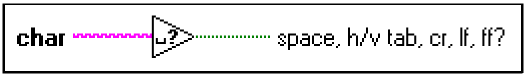
Select

Returns the value connected to the **t** input or **f** input, depending on the value of **s**. If **s** is TRUE, this function returns the value connected to **t**. If s is FALSE, this function returns the value connected to **f**.



White Space?

Returns TRUE if **char** is a white space character, such as space, Tab, Newline, Carriage Return, Form Feed, or Vertical Tab. Otherwise, the function returns FALSE.

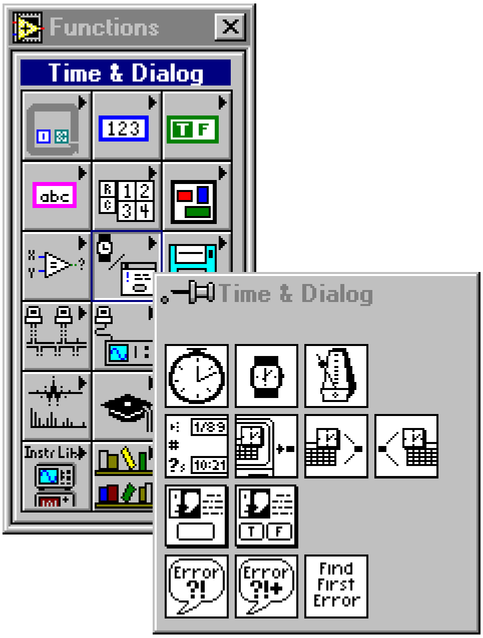


Chapter 10

Time, Dialog, and Error Functions

This chapter describes the timing functions, which you can use to get the current time, measure elapsed time, or suspend an operation for a specific period of time. Error Handling also is covered in this chapter.

The following illustration shows the **Time & Dialog** palette, which you access by selecting **Functions»Time & Dialog**.



For examples of time and dialog functions, see `examples\general\viopts.llb`.

Time, Dialog, and Error Functions Overview

This section introduces the Timing, Dialog, and Error Functions.

Timing Functions

The Date/Time To Seconds and the Seconds To Date/Time functions have a parameter called **date time rec,** which is a cluster that consists of signed 32-bit integers in the following order.

Table 10-1. Order of 32-bit Integers in TIming Functions

	Time	Value and Range
0	(second)	0 to 59
1	(minute)	0 to 59
2	(hour)	0 to 23
3	(day of month)	1 to 31 as output from the function;1 to 366 as input
4	(month)	1 to 12
5	(year)	1904 to 2040
6	(day of week)	1 to 7 (Sunday to Saturday)
7	(day of year)	1 to 366
8	(DST)	0 to 1 (0 for Standard Time, 1 for Daylight Savings Time)

The Wait (ms) and Wait Until Next ms Multiple functions make asynchronous system calls, but the nodes themselves function synchronously. That is, they do not complete execution until the specified time has elapsed. The functions use asynchronous calls so that other nodes can execute while the timing nodes wait.

☞ **Note:** ***National Instruments can only guarantee correct time values across all platforms for the range 2082844800 to 4230328447 seconds or 12:00 a.m., Jan. 1, 1970, Universal Time to 3:14 a.m., Jan. 19, 2038, Universal Time.***

Error Handling Overview

Every time you design a program, you should consider the possibility that something can go wrong and, if it does, you should consider how your program should manage the problem. LabVIEW automatically notifies you with a dialog box only when a few run-time errors occur, mostly for file dialog operations. It does not report all errors. If it were to report all errors, you would lose the flexibility to determine what to do when an error occurs and how and when to inform the user of the error in your program.

Rigorous error checking, especially for I/O operations (file, serial, GPIB, data acquisition, and communication), is invaluable in all phases of a project. This section describes three I/O situations in which errors can occur.

The first error can occur when you have initialized your communications incorrectly or have written improper data to your external device. This type of problem usually occurs during program development and disappears once you finish debugging your program. However, you can spend a lot of time tracking down a simple programming mistake because you have not incorporated error checks. Without error checks, all you know is that your program does not work. You do not know why the error occurred or where it is.

The second type of error can occur because your external device may be powered off, broken down, or otherwise unable to do what it normally does. This type of problem can occur at any time, but if you have incorporated error checking, your program notifies you immediately when such operational failures occur.

The third kind of error can occur when you upgrade LabVIEW or your operating system software, and you notice a bug in either a G program or a system program. This type of error means you should check errors that you may have felt safe ignoring, such as those from functions that close files or clear DAQ operations. The bottom line is, check all I/O operations for errors.

It may feel easier to ignore error checking when you have to add error handling code to test and report errors. The VIs described here are designed to make it easier for you to create programs with error checking and handling.

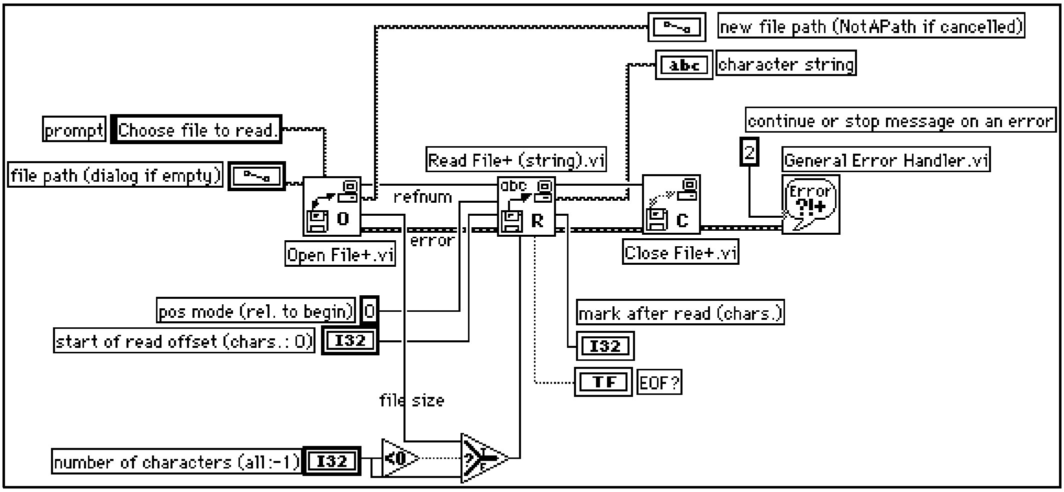
G functions and library VIs return errors in one of two ways—with numeric error codes or with an error state cluster. Typically, functions output error codes while VIs incorporate the error cluster, usually within a framework called error input/output or error I/O.

Error I/O and the Error State Cluster

The concept of error I/O is natural to the G dataflow architecture. If data information can flow from one node to another, so can error state information. Each node that needs to know about errors tests the incoming error state and responds appropriately. If no error exists, the node executes normally. If an error does exist, the node detects an error, skips execution, and then passes its error state out to the next node, which responds in the same way. In this fashion, notice of the first error that occurs in a sequence of operations is passed through all the nodes, with each node responding to the error. At the end of the flow, your program reports the error to the user.

Error I/O has an additional benefit—you can use it to control the execution order of independent operations. While you can use the DAQ taskID to control the order of DAQ operations for one group, you cannot use it to control the order for multiple groups. The DAQ taskID does not work with other types of I/O operations such as file operations.

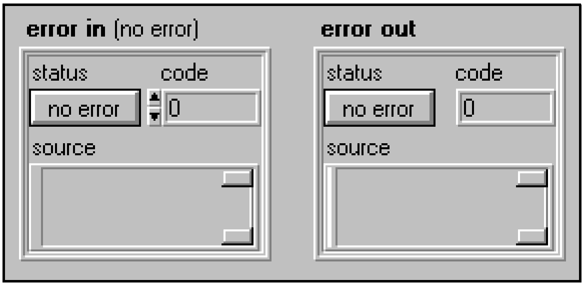
The following diagram from the File Utility VI, `Read Characters From File`, shows how error I/O is implemented in a simple VI.



The operation starts at `Open File+.vi`. If it opens the file successfully, `Read File+ (string).vi` reads the file and `Close File+.vi` closes the file. If you pass in an invalid path, `Open File+.vi` detects the error and passes the error state through the other two VIs to the General Error Handler, which reports it. Notice that the only presence of error handling on this block diagram is the error wire and the General Error Handler. It is neither cumbersome nor distracting.

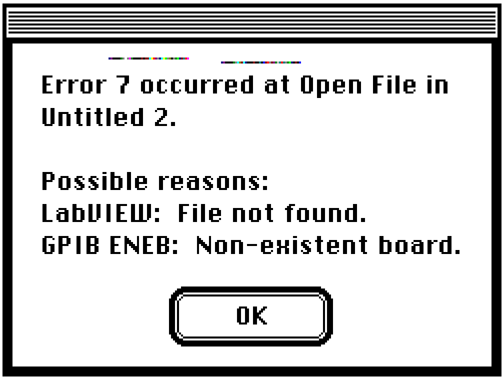
The error state consists of three pieces of information, which are combined into the error cluster. The **status** is a Boolean value—TRUE if an error exists, FALSE if it does not. The **code** consists of an unsigned 32-bit integer that identifies the error. In some cases, a non-zero error **code** coupled with a FALSE error **status** signals a warning rather than a fatal error. For example, a DAQ timeout event (code 10800) is typically reported as a warning. The **source** consists of a string that identifies where the error occurred.

The **error in** and **error out** state clusters for the `Open File+` VI, where the error shown in the preceding example originated, are shown in the following illustration. The **error in** cluster, whose default value is *no error* does not need to be wired if it is the first in the chain.



You can find the **error in** and **error out** clusters by selecting **Controls»Array & Cluster** on the front panel.

The following illustration shows the message you receive from the General Error Handler if you pass in an invalid path.



Error 7 occurred at Open File in Untitled 2.

Possible reasons:
LabVIEW: File not found.
GPIB ENEB: Non-existent board.

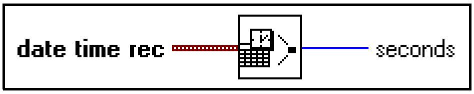
General Error Handler is one of the three error handling utility VIs. It contains a database of error codes and descriptions, from which it creates messages like the previous one. The Simple Error Handler performs the same basic operation but has fewer options. The third VI, Find First Error, creates the error I/O cluster from functions or VIs that output only scalar error codes.

Time and Dialog Function Descriptions

The following Time and Dialog functions are available.

Date/Time To Seconds

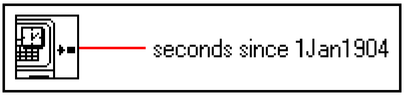
Converts a cluster of nine, signed 32-bit integers assumed to specify the local time (second, minute, hour, day, month, year, day of the week, day of the year, and Standard or Daylight Savings Time) in the configured time zone for your computer into a time-zone-independent number of **seconds** that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time.



If the year and month integers are out of range, the results are unpredictable. G ignores the day of the week and day of the year integers. The other five integers can be any value. Thus, you can specify Julian dates by setting the month to January and the current day to the day of the year. For example, use January 150 for the 150th day of the year.

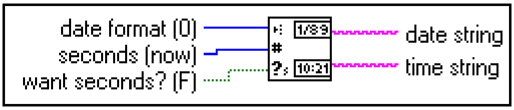
Get Date/Time In Seconds

Returns a time-zone-independent number that contains the number of seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time.



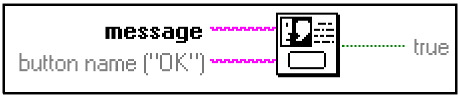
Get Date/Time String

Converts a time-zone-independent number assumed to be the number of seconds that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time, to a date and time string in the configured time zone for your computer.



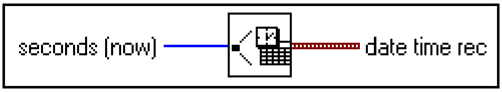
One Button Dialog Box

Displays a dialog box that contains a message and a single button. The **button name** is the name displayed on the dialog box button.



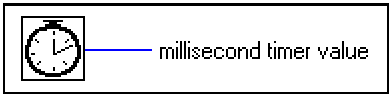
Seconds To Date/Time

Converts a time-zone-independent number assumed to be the number of **seconds** that have elapsed since 12:00 a.m., Friday, January 1, 1904, Universal Time, to a cluster of nine, signed 32-bit integers that specify (second, minute, hour, day of the month, number of month (1–12), year, day of the week, day of the year, and Standard or Daylight Savings Time) in the configured time zone for your computer.



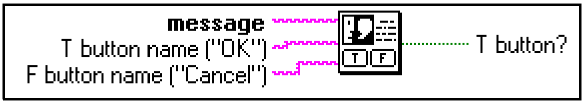
Tick Count (ms)

Returns the value of the millisecond timer. The base reference time (millisecond zero) is undefined; that is, you cannot convert **millisecond timer value** to a real-world time or date. Be careful when you use this function in comparisons, because the value of the millisecond timer wraps from $2^{32}-1$ to 0.



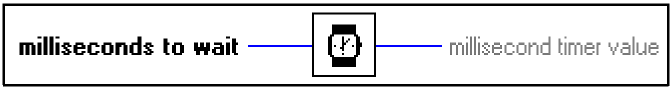
Two Button Dialog Box

Displays a dialog box that contains a **message** and two buttons. **T button name** and **F button name** are the names displayed on the buttons of the dialog box.



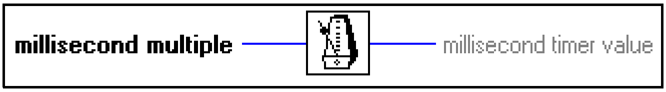
Wait (ms)

Waits the specified number of milliseconds and then returns the value of the millisecond timer.



Wait Until Next ms Multiple

Waits until the value of the millisecond timer becomes a multiple of the specified **millisecond multiple**. You can use this function to synchronize activities. You can call this function in a loop to control the loop execution rate. However, it is possible that the first loop period may be short.

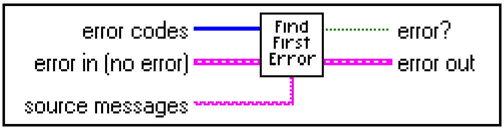


Error Handling VI Descriptions

The following Error Handling VIs are available.

Find First Error

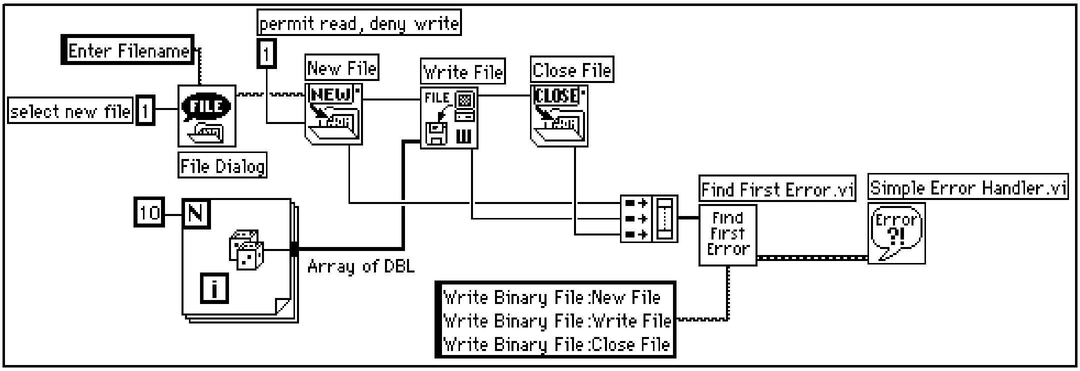
Tests the error status of one or more low-level functions or subVIs that output a numeric error code.



If this VI finds an error, it sets the parameters in the **error out** cluster. You can wire this cluster to the Simple or General Error Handler to identify the error and describe it to the user.

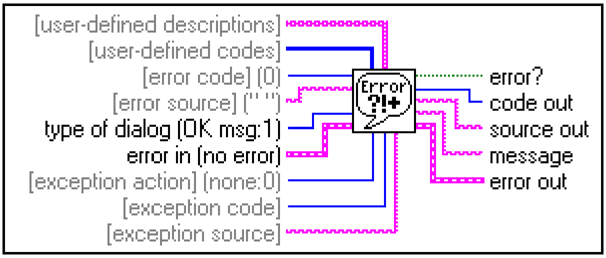
Find First Error Example

The following illustration shows how you can use Find First Error in the example VI Write Binary File. Find First Error creates the error cluster from individual error numbers, and Simple Error Handler reports any errors to the user.



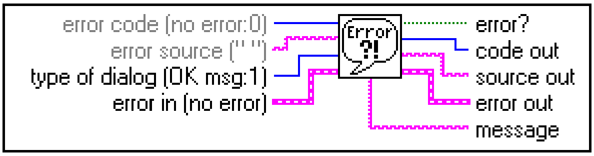
General Error Handler

Determines whether an error has occurred. If an error occurred, this VI creates a description of the error and optionally displays a dialog box.



Simple Error Handler

Determines whether an error occurred. If it finds an error, this VI creates a description of the error and optionally displays a dialog box.



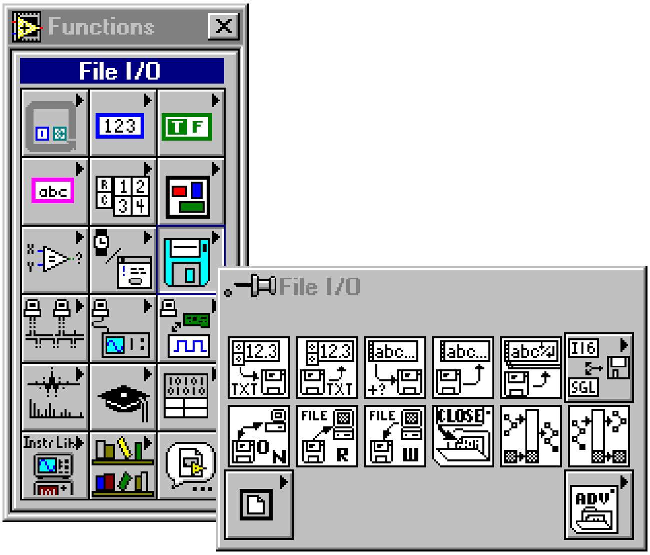
Simple Error Handler calls General Error Handler and has the same basic functionality as General Error Handler, but with fewer options.

Chapter 11

File Functions

This topic describes the low-level functions that manipulate files and directories. This topic also describes file constants and the high-level file VIs.

You access these functions, constants, and VIs by selecting **Functions»File I/O**.



The File I/O palette includes the following subpalettes:

- Advanced File Functions
- Binary File VIs
- File Constants

For examples of File functions and VIs, see `examples\file`.

File I/O VI and Function Overview

This section introduces the high-level and low-level File VIs, and the File functions.

High-Level VIs

You can use the high-level File VIs to write or read the following types of data:

- Strings to text files
- One-dimensional (1D) or two-dimensional (2D) arrays of single-precision numbers to spreadsheet text files.
- 1D or 2D arrays of single-precision or signed word integers to byte stream files.

The high-level File VIs described here call the low-level functions to perform complete, easy-to-use file operations. These VIs open or create a file, write or read to it, and close it. If an error occurs, these VIs display a dialog box that describes the problem and gives you the option to halt execution or to continue.

The high-level File VIs are located on the top row of the palette and consist of the following VIs:

- Write Characters to File
- Write to Spreadsheet File
- Read Characters from File
- Read from Spreadsheet File
- Read Lines from File
- Binary File VIs—located in the subpalette.

Low-Level File VIs and File Functions

The low-level File functions perform one file operation at a time. These VIs and functions perform error detection in addition to their other functions. The most commonly used low-level file functions and VIs are located on the second row of the palette. The remaining low-level functions are located in the **Advanced File Functions** subpalette.

The principal low-level file operations involve a three-step process. First, you create or open a file. Then you write data to the file or read data from the file. Finally, you close the file. Other file operations include creating directories; moving, copying, or deleting files; flushing files; listing directory contents; changing file characteristics; and manipulating paths.

When creating or opening a file, you must specify its location. Different computers describe the location of files in different ways, but most computer systems use a hierarchical system to specify the location of files. In a hierarchical file system, the computer system superimposes a hierarchy on the storage media. You can store files inside directories, which can contain other directories.

When you specify a file or directory in a hierarchical file system, you must indicate the name of the file or directory, as well as its location in the hierarchy. In addition, some file systems support the connection of multiple discrete media, called volumes. For example, Windows systems support multiple drives connected to a system; for most of these systems, you must include the name of the volume to create a complete specification for the location of a file. On other systems, such as UNIX, you do not need to specify the storage media locations for files because the operating system hides the physical implementation of the file system from you.

The method of identifying the target of a file function varies depending on whether the target is an open file. If the target is not an open file, or if it is a directory, you specify a target using the *path* of the target. The path describes the volume containing the target, the directories between the top-level and the target, and the name of the target. If the target is an open file, you use a *file refnum* to identify the file that G is supposed to manipulate. The file refnum is an identifier that G associates with the file when you open it. When you close the file, the file manager dissociates the file refnum from the file. In other words, the refnum is obsolete once the file is closed.

See, *Strings and File I/O*, Chapter 6 of the *Tutorial Manual*, and *Path Controls and Refnum* in that section for more information on path specification in G and for file function examples.

Byte Stream and Datalog Files

G can make and access two types of files—byte stream and datalog files.

A *byte stream* file, as the name implies, is a file whose fundamental unit is a byte. A byte stream file can contain anything from a homogeneous set of one G datatype to an arbitrary collection of datatypes—characters, numbers, Booleans, arrays, strings, clusters, and so on. An ASCII text file, a file containing this paragraph, for example, is perhaps the simplest byte stream file. A similar byte stream file is a basic spreadsheet text file, which consists of rows of ASCII numbers, with the numbers separated by tabs and the rows separated by carriage returns.

Another simple byte stream file is an array of binary 16-bit integers or single-precision, floating point numbers, which you acquire from a data acquisition (DAQ) program. A more complicated byte stream file is one in which an array of binary 16-bit integers or single-precision, floating point numbers is preceded by a header of ASCII text that describes how and when you acquired the data. That header could alternatively be a cluster of acquisition parameters, such as arrays of channels and scale factors, the scan rate, and so forth.

An Excel worksheet file, as opposed to an Excel text file, is also a more complicated form of byte stream file because it contains text interspersed with Excel-specific formatting data that does not make sense when you read it as text. In summary, you can make a byte stream file that consists of one each of all of G datatypes. Byte stream files can be created using high-level VIs and low-level functions.

A *datalog* file, on the other hand, consists of a sequence of identically-structured records. Like byte stream files, the components of a datalog record can be any G datatype. The difference is that all the datalog records must be the same type. Datalog files can only be created using low-level file functions.

You write a byte stream file typically by appending new strings, numbers, or arrays of numbers of any length to the file. You can also overwrite data anywhere within the file. You write a datalog file by appending one record at a time. You cannot overwrite the record.

You read a byte stream file by specifying the byte offset or index and the number of instances of the specified byte stream type you want to read. You read a datalog file by specifying the record offset or index and the number of records you want to read.

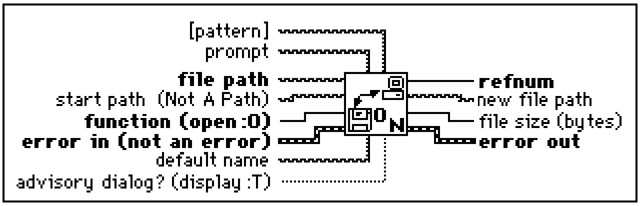
You use byte stream files typically for text or spreadsheet data that other applications may need to read. You can use byte stream files to record continuously acquired data that you need to read sequentially or randomly in arbitrary amounts. You use datalog files typically to record multiple test results or waveforms that you read one at a time and treat individually. Datalog files are difficult to read from non-G applications.

Flow-Through Parameters

Many file functions contain *flow-through* parameters, which return the same value as an input parameter. You can use these parameters to control the execution order of the functions. By wiring the flow-through output of the first node you want to execute to the corresponding input of the next node you want to execute, you create artificial data dependency. Without these flow-through parameters, you would often have to use Sequence structures to ensure that file I/O operations take place in the correct order.

Error I/O in File I/O Functions

G uses error I/O clusters, consisting of **error in** and **error out**, in all of its file I/O functions. With error I/O clusters you can string together several functions. When an error occurs in a function, that function passes the error along to the next function. When the error passes to subsequent functions, the subsequent function does not execute and passes the error along to the following function, and so on. The following illustration displays an example of the **error in** and **error out** clusters.



Although the error I/O clusters specify whether an error has occurred, you may want to use error handlers to report the error to the user. For more information on error I/O, see Chapter 10, *Time, Dialog, and Error Functions*, in this manual.

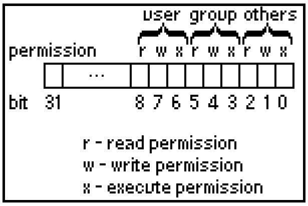
Permissions

Some of the File Functions have a 32-bit integer parameter called **permissions** or **new permissions**. G uses only the least significant nine bits of the 32-bit integer to determine file and directory access permissions.

(Windows) G ignores the permissions for directories. For files, only bit 7 (the UNIX user write permission bit) is used. If this bit is clear, the file is read-only. Otherwise, you can write to the file.

(Macintosh) G uses all 9 bits of permissions for directories. The bits which control read, write, and execute permissions, respectively, on a UNIX system are used to control See Files, Make Changes, and See Folders access rights, respectively, on the Macintosh. For files, only bit 7 (the UNIX user write permission bit) is used. If this bit is clear, the file is locked. Otherwise, the file is not locked.

(UNIX) The nine bits of permissions correspond exactly to nine UNIX permission bits governing read, write, and execute permissions for users, groups, and others. The following illustration shows the permission bits on a UNIX system.

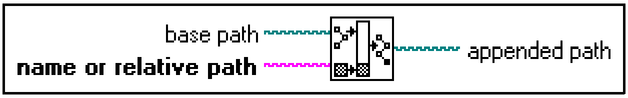


File I/O Function and VI Descriptions

The following functions and VIs are available from the File I/O palette.

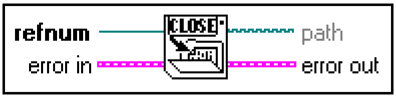
Build Path

Creates a new path by appending a name (or relative path) to an existing path.



Close File

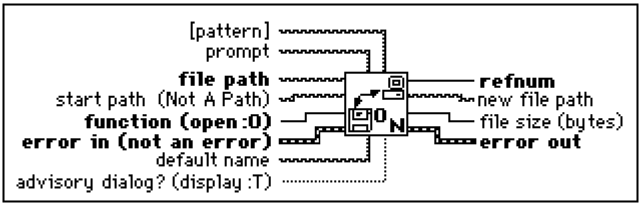
Writes all buffers of the file identified by **refnum** to disk, updates the directory entry of the file, closes the file, and voids **refnum** for subsequent file operations.



☞ **Note:** ***Error I/O functions uniquely in the Close File function, which closes regardless of whether an error occurred in a preceding operation, insuring that files are closed correctly.***

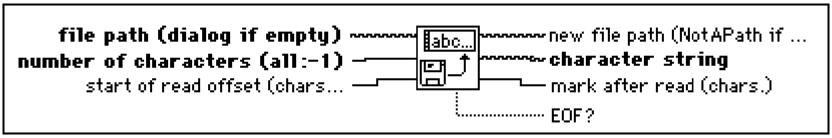
Open/Create/Replace File

Opens an existing file, creates a new file, or replaces an existing file, programmatically or interactively using a file dialog box. You can optionally specify a dialog **prompt**, default file name, **start path**, or filter **pattern**. Use this VI with the intermediate Write File or Read File functions.



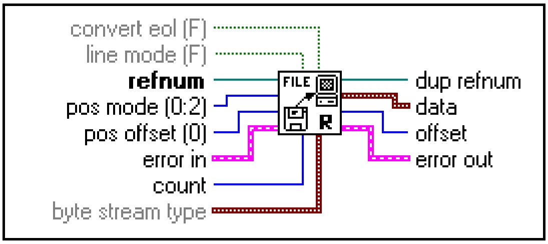
Read Characters From File

Reads a specified number of characters from a byte stream file beginning at a specified character offset. The VI opens the file before reading from it and closes it afterwards.


file path (dialog if empty)
number of characters (all:-1)
start of read offset (chars...
abc...
new file path (NotAPath if ...
character string
mark after read (chars.)
EOF?

Read File

Reads data from the file specified by **refnum** and returns it in **data**. Reading begins at a location specified by **pos mode** and **pos offset** and depends on the format of the specified file.



Reading Byte Stream Files

If **refnum** is a byte stream file refnum, the Read File function reads data from the byte stream file specified by **refnum**. You can wire either **line mode** or **byte stream type** when you read byte stream files, but you cannot wire both. If you do not wire **byte stream type**, Read File assumes the data that begins at the designated byte offset is a string of characters. If you wire **byte stream type**, the function interprets **data** starting at the designated byte offset to be **count** instances of that type. Following the read operation, the function sets the file mark to the byte following the last byte read. If the function encounters end of file before reading all of the requested data, it returns as many whole instances of the designated **byte stream type** as it finds.

Reading Characters

To read characters from a byte stream file (typically a text file) do not wire the **byte stream type**. The following paragraphs describe the manner in which the **line mode**, **count**, **convert eol**, and **data** parameters function when reading from a byte stream file.

line mode, in conjunction with **count**, determines when the read stops.

If **line mode** is TRUE, and if you do not wire **count** or **count** equals 0, Read File reads until it encounters an end of line marker—a carriage return, a line feed, or a carriage return followed by a line feed, or it encounters end of file. If **line mode** is TRUE, and **count** is greater than 0, Read File reads until it encounters an end of line marker, it encounters end of file, or it reads **count** characters.

If **line mode** is FALSE, Read File reads **count** characters. In this case, if you do not wire **count**, it defaults to 0. **line mode** defaults to FALSE.

convert eol (F) determines whether the function converts the end of line markers it reads into G end of line markers. The system-specific end of line marker is a carriage return followed by a line feed on Windows, a carriage return on Macintosh, and a line feed on UNIX. The G end of line marker is a line feed.

If **convert eol** is TRUE, the function converts all end of line markers it encounters into line feeds. If **convert eol** is FALSE, the function does not convert the end of line markers it reads. **convert eol** defaults to FALSE.

data is the string of characters read from the file.

Reading Binary Data

To read binary data from a byte stream file, wire the type of the data to **byte stream type**. In this case, **count**, and **data** function in the manner described in the following paragraphs, and you do not have to wire **line mode** or **convert eol**.

byte stream type can be any datatype. Read File interprets the data starting at the designated byte offset to be **count** instances of that type. If the type is variable-length, that is, an array, a string, or a cluster containing an array or string, the function assumes that each instance of the type contains the length or dimensions of that instance. If they do not, the function misinterprets the data. If G determines that the data does not match the type, it sets the value of **data** to the default value for its type and returns an error.

count is the number of instances of the **byte stream type** to read. If **count** is unwired, the function returns a single instance of the **byte stream type**.

If you wire **count**, it can be a scalar number, in which case the function returns a 1-D array of instances of the **byte stream type**. Or it can be a cluster of N scalar numbers, in which case the function returns an N-dimension array of instances of the **byte stream type**.

If the wired **count** is a scalar number and the **byte stream type** is something other than an array, the function returns that number of instances in a 1D array. For example, if the type is a single-precision, floating point number, the function returns an array of three, single-precision, floating point numbers. However, if the type is an array, the function returns the instances in a cluster array, because G does not have arrays of arrays. Therefore, if the type is an array of single-precision, floating point numbers and **count** is 3, the function returns a cluster array of three, single-precision, floating point number arrays.

If the wired **count** is a cluster of N numbers, the function returns an N-dimension array of instances of the type. The size of each dimension is the value of the corresponding number according to its cluster order. The number of instances returned in this manner is the product of the N numbers. Thus, you can return 20, single-precision, floating point numbers as a 2D array of two columns and ten rows by wiring a two-element cluster with element 0 = 2 and element 1 = 10 to **count**.

data contains the data read from the file. Refer to the previous description of **count** for an explanation of the structures data can have.

Reading Datalog Files

If **refnum** is a datalog file refnum, the Read File function reads records from the datalog file specified by **refnum**. If the data in the file does not match the datatype associated with the datalog file, this function returns an error.

The number of records read can be less than specified by **count** if this function encounters the end of the file. The function sets the file mark to the record following the last record read. (You should never encounter a partial record; if you do, the file is corrupt.)

Do not wire **convert eol**, **line mode**, and **byte stream type**. They do not pertain to datalog files. The **count** and **data** parameters function in the following manner.

count is the number of records to read and may be wired or unwired. If you do not wire **count**, the function returns a single record of the datalog type specified when the file is created or opened. For example, if the type is a 16-bit integer, the function returns one 16-bit integer. If the type is an array of 16-bit integers, the functions returns one array of 16-bit integers. (Your records typically consist of clusters of diverse elements, but the rules for simple types used in these examples apply to those as well.)

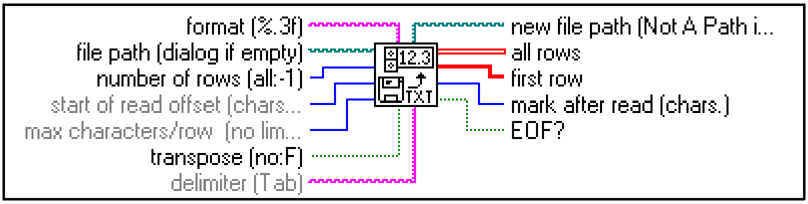
If you wire **count**, it can be a scalar number, in which case the function returns a 1D array of records. Or it can be a cluster of N scalar numbers, in which case the function returns an N-dimension array of records.

If the wired **count** is a scalar number, and the datalog type is something other than an array, the function returns that number of records in a 1D array. For example, if the type is a single-precision, floating-point number and **count** is 3, the array contains three, single-precision, floating-point numbers. However, if the type is an array, the function returns the records in a cluster array (because G does not have arrays of arrays). Therefore, if the datalog type is an array of single-precision, floating-point numbers and **count** is 3, the function returns a cluster array of three, single-precision, floating-point number arrays.

If the wired **count** is a cluster of N numbers, the function returns an N-dimension array of records. The size of each dimension is the value of the corresponding number according to its cluster order. The number of records returned in this manner is the product of the N numbers. Therefore, you can return 20 records as a 2D array of two columns and ten rows by wiring a two-element cluster with element 0 = 2 and element 1 = 10 to **count**.

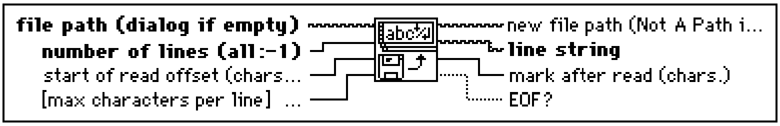
Read From Spreadsheet File

Reads a specified number of lines or rows from a numeric text file beginning at a specified character offset and converts the data to a 2D, single-precision array of numbers. Optionally, you can transpose the array. The VI opens the file before reading from it and closes it afterwards. You can use this VI to read a spreadsheet file saved in text format. This VI calls the Spreadsheet String to Array function to convert the data.



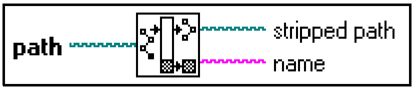
Read Lines From File

Reads a specified number of lines from a byte stream file beginning at a specified character offset. The VI opens the file before reading from it and closes it afterwards.


file path (dialog if empty)
number of lines (all:-1)
start of read offset (chars...
[max characters per line] ...
abc
new file path (Not A Path i...
line string
mark after read (chars.)
EOF?

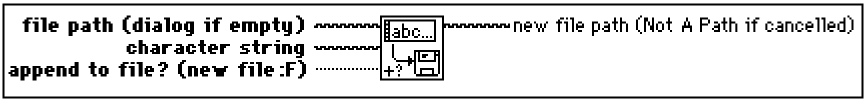
Strip Path

Returns the **name** of the last component of a path and the **stripped path** that leads to that component.



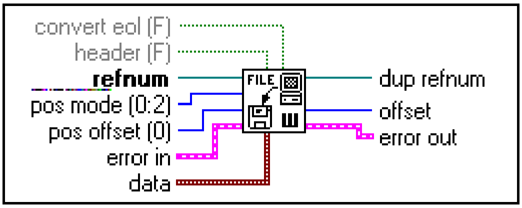
Write Characters To File

Writes a character string to a new byte stream file or appends the string to an existing file. The VI opens or creates the file before writing to it and closes it afterwards.


file path (dialog if empty)
character string
append to file? (new file:F)
abc...
+?
new file path (Not A Path if cancelled)

Write File

Writes data to the file specified by **refnum**. Writing begins at a location specified by **pos mode** and **pos offset** for byte stream file and at the end of file for datalog files. **data**, **header**, and the format of the specified file determine the amount of data written.



Writing Byte Stream Files

If **refnum** is a byte stream file refnum, the Write File function writes to a location specified by **pos mode** and **pos offset** in the byte stream file specified by **refnum**. If the top-level datatype of **data** is of variable length (that is, a string or an array), Write File can write a **header** to the file that specifies the size of the data. G sets the file mark to the byte following the last byte written. **convert eol** determines whether the function converts the end-of-line markers it writes into system-specific end-of-line markers. You can wire **convert eol** only if **data** is a string. The system-specific end-of-line marker is a carriage return followed by a line feed on Windows, a line feed on UNIX, and a carriage return on Macintosh. If **header** is true, G ignores **convert eol**.

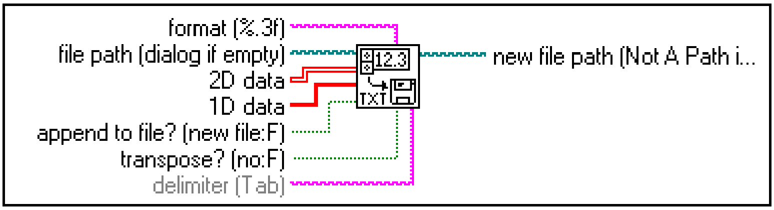
Writing Datalog Files

If **refnum** is a datalog file refnum, the Write File function writes data as records to the datalog file specified by **refnum**. Writing always starts at the end of the datalog file (datalog files are append-only). G sets the file mark to the record following the last record written. The **convert eol**, **header**, **pos mode**, and **pos offset** parameters do not apply with datalog files, and you cannot wire them. The **data** parameter functions in the following manner for datalog files.

data must be either a datatype that matches the datatype specified when you open or create the file, or an array of such datatypes. In the former case, this function writes **data** as a single record in the datalog file. Representation of numeric data is coerced to the representation of the datatype if necessary. In the latter case, this function writes each element of **data** as a separate record in the datalog file in row-major order.

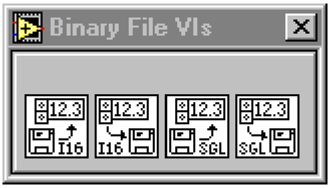
Write To Spreadsheet File

Converts a 2D or 1D array of single-precision (SGL) numbers to a text string and writes the string to a new byte stream file or appends the string to an existing file. You can optionally transpose the data. This VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to create a text file readable by most spreadsheet applications. This VI calls the Array to Spreadsheet String function to convert the data.


format (%.3f)
file path (dialog if empty)
2D data
1D data
append to file? (new file:F)
transpose? (no:F)
delimiter (Tab)
12.3
TXT
new file path (Not A Path i...

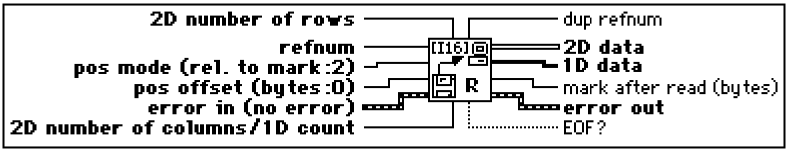
Binary File VI Descriptions

The following VIs are available on the **Binary File VIs** subpalette.



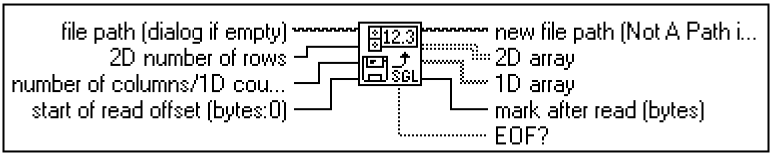
Read From I16 File

Reads a 2D or 1D array of data from a byte stream file of signed, word integers (I16). The VI opens the file before reading from it and closes it afterwards. You can use this VI to read unscaled or binary data acquired from data acquisition VIs and written to a file with Write To I16 File.



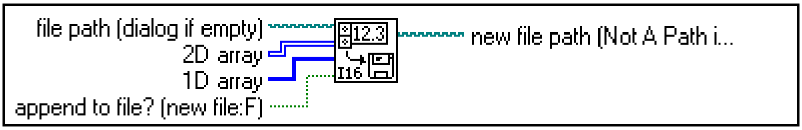
Read From SGL File

Reads a 2D or 1D array of data from a byte stream file of single-precision numbers (SGL). The VI opens the file before reading from it and closes it afterwards. You can use this VI to read scaled data acquired from data acquisition VIs and written to a file with Write To SGL File.


file path (dialog if empty)
2D number of rows
number of columns/1D cou...
start of read offset (bytes:0)
12.3
SGL
new file path (Not A Path i...
2D array
1D array
mark after read (bytes)
EOF?

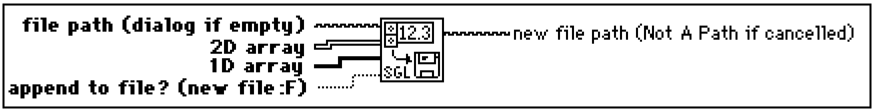
Write To I16 File

Writes a 2D or 1D array of signed word integers (I16) to a new byte stream file or appends the data to an existing file. The VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to write unscaled or binary data from data acquisition VIs.


file path (dialog if empty)
2D array
1D array
append to file? (new file:F)
new file path (Not A Path i...
12.3
I16

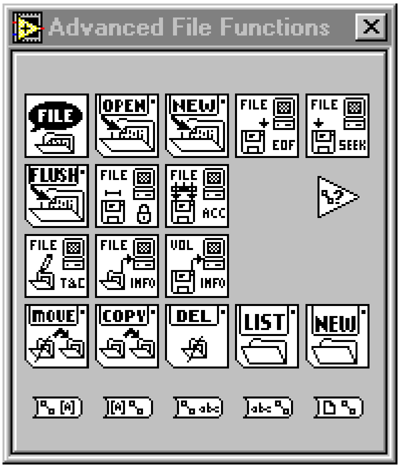
Write To SGL File

Writes a 2D or 1D array of single-precision numbers (SGL) to a new byte stream file or appends the data to an existing file. The VI opens or creates the file before writing to it and closes it afterwards. You can use this VI to write scaled data from data acquisition VIs without changing the representation.


file path (dialog if empty)
2D array
1D array
append to file? (new file:F)
new file path (Not A Path if cancelled)
12.3
SGL

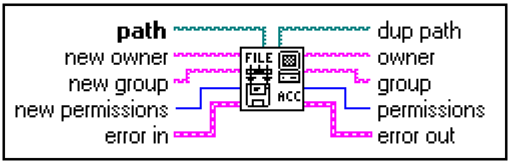
Advanced File Function Descriptions

The following functions are available on the **Advanced File Functions** subpalette.



Access Rights

Sets and returns the owner, group, and permissions of the file or directory specified by **path**. If you do not specify **new owner**, **new group**, or **new permissions**, this function returns the current settings unchanged.

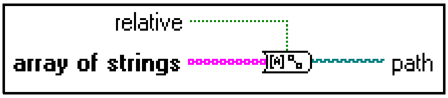


(Windows) The Access Rights function ignores **new owner** and **new group** and returns empty strings for **owner** and **group** because Windows does not support owners and groups.

(Macintosh) If path refers to a file, the Access Rights function ignores **new owner** and **new group** and returns empty strings for **owner** and **group** because Macintosh does not support owners or groups for files.

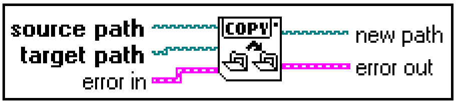
Array Of Strings To Path

Converts an **array of strings** into a relative or absolute **path**.



Copy

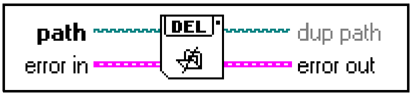
Copies the file or directory specified by **source path** to the location specified by **target path**. If you copy a directory, this function copies all its contents recursively.



Delete

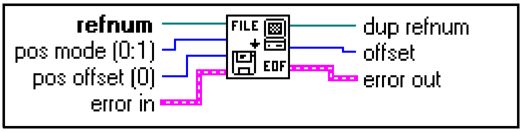
Deletes the file or directory specified by **path**. If **path** specifies a directory that is not empty or if you do not have write permission for both the file or directory specified by

path and its parent directory, this function does not remove the directory and returns an error.



EOF

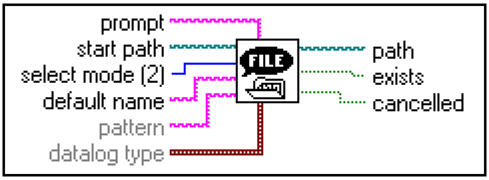
Sets and returns the logical EOF (end-of-file) of the file identified by **refnum**. **pos mode** and **pos offset** specify the new location of the EOF. If you do not specify pos mode or pos offset, this function returns the current unchanged EOF. This function always returns the location of the EOF relative to the beginning of the file.



You cannot set the EOF of a datalog file. If **refnum** identifies a datalog file, you cannot wire **pos mode** and **pos offset**. However, you still can get the EOF of a datalog file, which tells you how many records exist in the file.

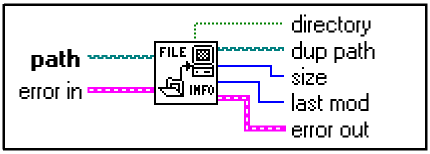
File Dialog

Displays a dialog box with which you can specify the path to a file or directory. You can use this dialog box to select existing files or directories or to select a location and name for a new file or directory.



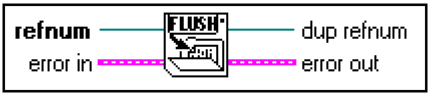
File/Directory Info

Returns information about the file or directory specified by **path**, including its **size**, its last modification date, and whether it is a directory.



Flush File

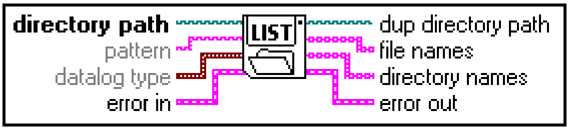
Writes all buffers of the file identified by **refnum** to disk and updates the directory entry of the file associated with **refnum**. The file remains open, and **refnum** remains valid.



Data written to a file often resides in a buffer until the buffer fills up or until you close the file. This function forces the operating system to write any buffer data to the file.

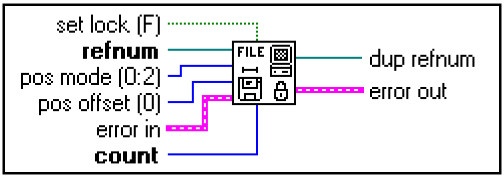
List Directory

Returns two arrays of strings listing the names of all files and directories found in **directory path**, filtering both arrays based upon **pattern** and filtering the **file names** array based upon the specified **datalog type**.



Lock Range

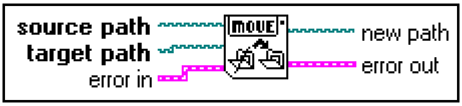
Locks or unlocks a range of a file specified by **refnum**. Locking a range of a file prevents both reading and writing by other users, overriding permissions for the file, and the deny mode associated with **refnum**. See *File I/O VI and Function Overview* earlier in this manual for a full discussion of permissions. Unlocking a range of a file removes the override caused by locking a range, so that the file's permissions and the deny mode associated with **refnum** determine whether other users can read from or write to that range of the file.



You cannot lock a range of a datalog file.

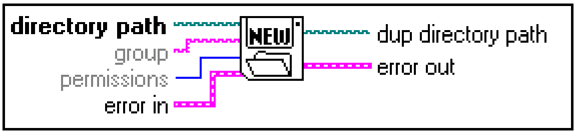
Move

Moves the file or directory specified by **source path** to the location specified by **target path**.



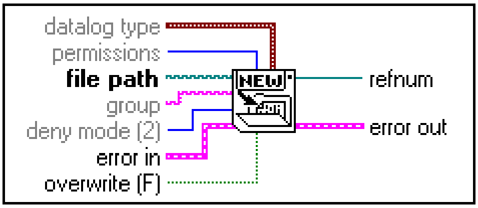
New Directory

Programmatically creates the directory specified by **directory path**. If a file or directory already exists at the specified location, this function returns an error instead of overwriting the existing file or directory.



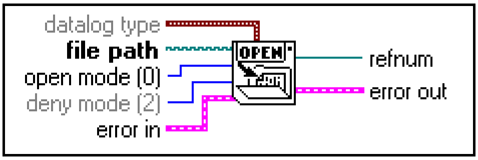
New File

Creates the file specified by **file path** and opens it for reading and writing (regardless of **permissions**).



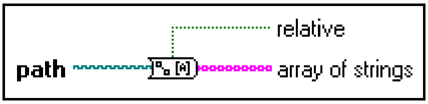
Open File

Opens the file specified by **file path** for reading and/or writing.



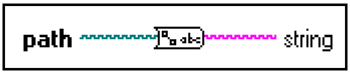
Path To Array Of Strings

Converts a **path** into an **array of strings** and indicates whether the path is **relative**.



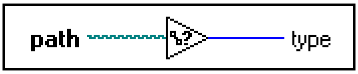
Path To String

Converts **path** into a string describing a path in the standard format of the platform.



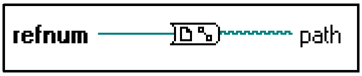
Path Type

Returns the type of the specified path, indicating whether it is an absolute, relative, or invalid path. This function checks only the format of the path, not whether the path refers to an existing file or directory. Therefore, this function only indicates an invalid path for Not A Path.



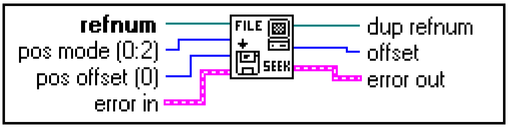
Refnum To Path

Returns the **path** associated with the specified **refnum.**



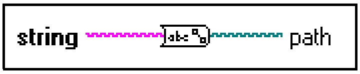
Seek

Moves the current file mark of the file identified by **refnum** to the position indicated by **pos offset** according to the mode chosen by **pos mode**.



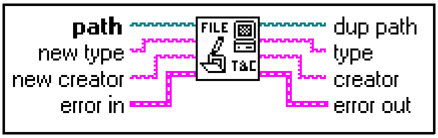
String To Path

Converts a **string**, describing a path in the standard format for the current platform, to a **path**.



Type and Creator

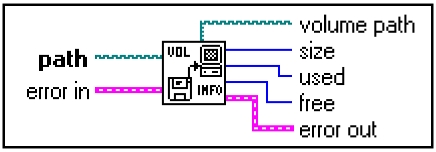
Reads and sets the type and creator of the file specified by **path**. File type and creator are four-character strings. If you do not specify **new type** or **new creator**, this function returns the current settings unchanged.



Windows and UNIX do not support file types and creators. Trying to set the type or creator of a file in these platforms results in an error; however, you can get the file type and creator in these platforms. If the specified file has a name ending with characters that LabVIEW recognizes as specifying a file type (such as .vi for the LVIN file type and `.llb` for the LVAR file type), this function returns that type in **type** and `LBVW` in **creator**. Otherwise, the function returns `????` in both **type** and **creator**.

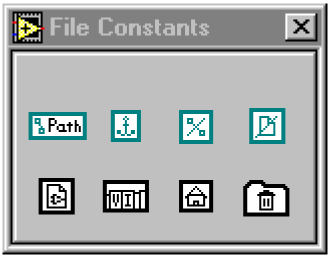
Volume Info

Returns information about the volume containing the file or directory specified by **path**, including the total storage space provided by the volume, the amount used, and the amount free in bytes.



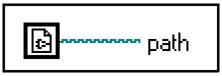
File Constants Descriptions

The following constants are the options available on the **File Constants** subpalette.



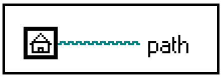
Current VI's Path Constant

Returns the path to the file containing the VI in which this function appears. If the VI is incorporated into an application (using the Application Builder libraries), the function returns the path to the VI in the application file, and treats the application file as a VI library.



Default Directory Constant

Returns the path to your default directory. The default directory is the directory which the file dialog displays initially. The G Preferences dialog box, under **Paths**, defines this directory.



Empty Path

Returns an empty path.

Not A Path

Returns a path whose value is Not A Path. You can use this path as an output from structures and subVIs when an error occurs.

Not A Refnum

Returns a refnum whose value is Not A Refnum. You can use this refnum as an output from structures and subVIs when an error occurs.

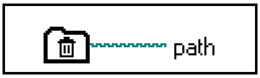
Path Constant

Use this to supply a constant directory or file path to the block diagram. Set this value by clicking inside the constant with the Operating tool and typing in the value. Use the standard file path syntax for a given platform.

The value of the path constant cannot be changed while the VI executes. You can assign a label to this constant.

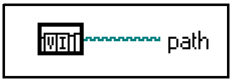
Temporary Directory Constant

Returns the path to your temporary directory. The temporary directory is the directory in which you store temporary information that you expect the user or the operating system to delete periodically. The **Preferences** dialog box, under **Paths**, defines this directory.



VI Library Constant

Returns the path to the VI library directory for the current G on the current computer. The G Preferences dialog box (**Edit»Preferences**) defines this directory. If you build an application using the Application Builder libraries, this path is the path of the directory containing the application.

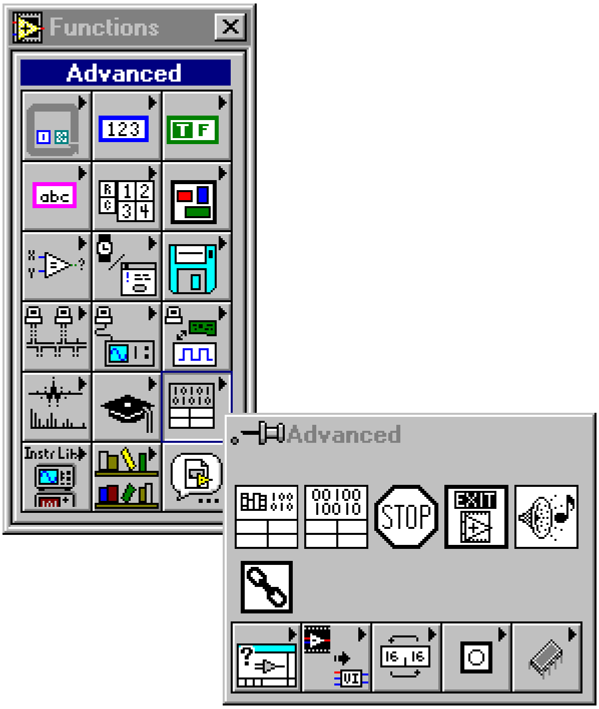


Chapter 12

Advanced Functions

This chapter describes the functions that perform advanced operations. This chapter also describes the Help, Data Manipulation, and Occurrence Functions, and the VI Control and Memory VISA.

To access the **Advanced** palette, shown in the following illustration, select **Functions»Advanced**.



The Advanced Functions include the following subpalettes:

- Data Manipulation
- Help
- Memory

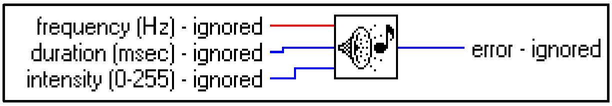
- Occurrences
- VI Controls

Advanced Function Descriptions

The following Advanced Functions are available.

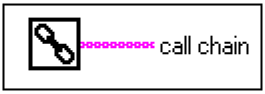
Beep

Causes the system to issue an audible tone. You can specify the tone frequency in Hertz, the duration in milliseconds, and the intensity as a value from 0 to 255, with 255 being the loudest. Although this VI appears on all platforms, the frequency, duration, and intensity parameters work only on the Macintosh.



Call Chain

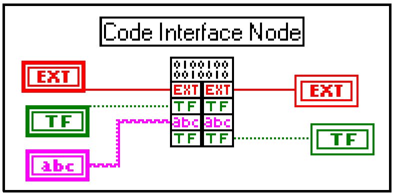
Returns the chain of callers from this VI to the top-level VI as an array of strings.



Code Interface Node

With a Code Interface Node (CIN), you can call code written in a conventional programming language, such as C, directly from a block diagram. CINs make it possible for you to use algorithms written in another language or to access platform-specific features or hardware that G does not directly support.

Code Interface Nodes are resizable and show datatypes for the connected inputs and outputs, similar to the Bundle function. The following illustration shows the CIN function.



LabVIEW's interface to external code is very powerful. You can pass any number of parameters to or from external code, and each parameter can be of any arbitrary G datatype. LabVIEW provides several libraries of routines that make working with G datatypes easier. These routines support memory allocation, file manipulation, and datatype conversion.

If you convert a VI that contains a CIN to another platform, you need to recompile the code for the new platform, because CINs use code compiled in another programming language. You can write source code for a CIN so that it is machine-independent, requiring only a recompile to convert it to another platform.

For examples of CINs, see `examples\cins`.

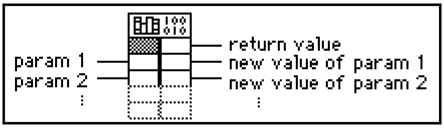
For more information on the Code Interface Node see the *Code Interface Reference Manual*.

Call Library Function

With the Call Library Function node, you can call standard libraries without writing a Code Interface Node (CIN). Under Windows, you can call a dynamic link library (DLL) function directly. In Macintosh and UNIX, you can call a shared library function directly. On the Macintosh 68K, you must have the CFM-68K system extension installed for the Call Library Function node to operate.

This node supports a large number of datatypes and calling conventions. You should be able to use it to call functions from most standard and custom-made libraries.

The Call Library Function node, shown in the following illustration, looks similar to a Code Interface Node.

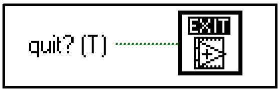


The Call Library Function consists of paired input/output terminals with input on the left and output on the right. You can use one or both. The return value for the function is returned in the right terminal of the top pair of terminals of the node. If there is no return value, then this pair of terminals is unused. Each additional pair of terminals corresponds to a parameter in the functions parameter list. You pass a value to the function by wiring to the left terminal of a terminal pair. You read the value of a parameter after the function call by wiring from the right terminal of a terminal pair.

If you select **Configure...** from the pop-up menu of the node, you see a Call Library Function dialog box from which you can specify the library name or path, function name, calling conventions, parameters, and return value for the node. When you click on **OK**, the node automatically increases in size to have the correct number of terminals. It then sets the terminals to the correct datatypes. For more information on Call Library Function refer to Chapter 24, *Calling Code From Other Languages*, in the *LabVIEW User Manual*.

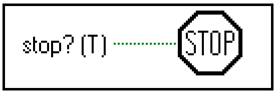
Quit

Stops all executing VIs and ends the current session of LabVIEW. This function shuts down only LabVIEW; the function does not power down the system or affect other applications. The function stops running VIs the same way the Stop function does.



Stop

Stops the VI in which it executes, just as if you clicked the stop button in the toolbar. If you wired the input, stop occurs only if the input value is TRUE. If you leave the input unwired, the stop occurs as soon as the node that is currently executing finishes.

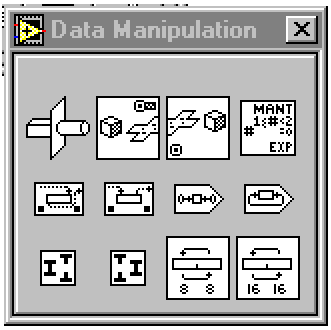


If you need to abort execution of all VIs in a hierarchy from the block diagram, you can use this function, but *you must use it with caution.* Before you call the Stop function with a TRUE input, be sure to complete all final tasks for the VI first, such as closing files, setting save values for devices being controlled, and so on. If you put the Stop function in a subVI, you should make its behavior clear to other users of the VI, because this function causes their VI hierarchies to abort execution.

In general, you should avoid using the Stop function when you have a built-in terminator protocol in your VI. For example, I/O operations should be performed in While Loops so that the VI can terminate the loop on an I/O error. You should also consider using a front panel Stop Boolean control to terminate the loop at the request of the user rather than using the Stop function.

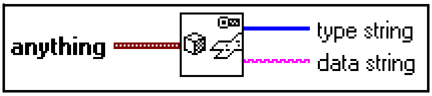
Data Manipulation Function Descriptions

The following illustration displays the options available on the **Data Manipulation** subpalette.



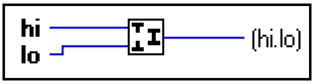
Flatten To String

Converts **anything** to a string of binary values. **type string** is a type descriptor that describes the datatype of anything. data string is the flattened form of anything. For more information on type descriptors and flattened data, see *Flattened Data*, in Appendix A, *Data Storage Formats*, of the *LabVIEW User Manual*.



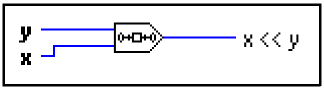
Join Numbers

Creates a number from the component bytes or words.



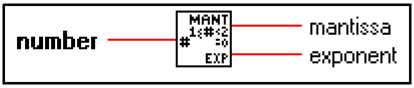
Logical Shift

Shifts **x** the number of bits specified by **y**.



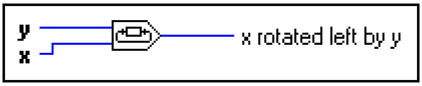
Mantissa & Exponent

Returns the mantissa and exponent of the input numeric value such that **number** = **mantissa** * $2^{\textbf{exponent}}$. If **number** is 0, both **mantissa** and **exponent** are 0. Otherwise, the value of **mantissa** is greater than or equal to 1 and less than 2, and the value of **exponent** is an integer.



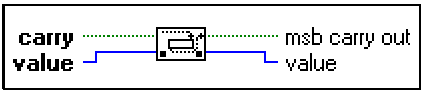
Rotate

Rotates **x** the number of bits specified by **y**.



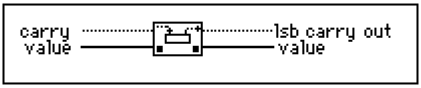
Rotate Left With Carry

Rotates each bit in the input **value** to the left (from least significant to most significant bit), inserts **carry** in the low-order bit, and returns the most significant bit.



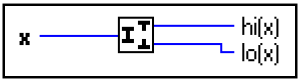
Rotate Right With Carry

Rotates each bit in **value** to the right (from most significant to least significant), inserts **carry** in the high-order bit, and returns the least significant bit.

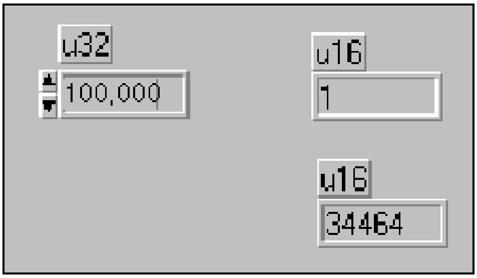


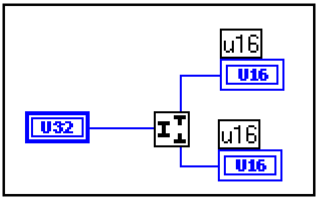
Split Number

Breaks a number into its component bytes or words.



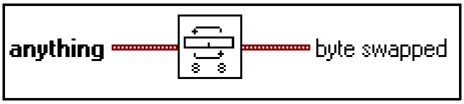
The following illustration shows an example of how to use the Split Number function. The function splits the signed 32-bit number 100,000 into the high word component, 1, and the low word component, 34,464.





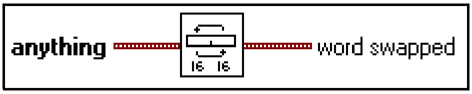
Swap Bytes

Swaps the high-order 8 bits and the low-order 8 bits for every word in **anything**.



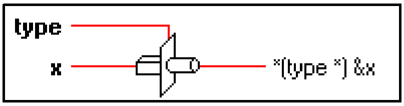
Swap Words

Swaps the high-order 16 bits and the low-order 16 bits for every long integer in **anything**.



Type Cast

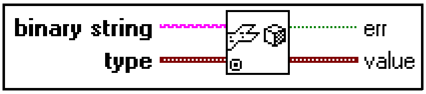
Casts **x** to the datatype, **type**.



Casting data to a string converts it into machine-independent, big endian form. That is, the function puts the most significant byte or word first and the least significant byte or word last, removes alignment, and converts extended-precision numbers to 16 bytes. Casting a string to a 1D array converts the string from machine-independent form to the native form for that platform.

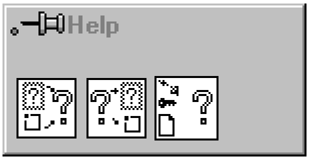
Unflatten From String

Converts **binary string** to the type wired to **type**. This function performs the inverse of Flatten To String. **binary string** should contain flattened data of the type wired to **type**. For more information on type descriptors and flattened data, see *Flattened Data*, in Appendix A, *Data Storage Formats*, of the *LabVIEW User Manual*.



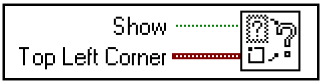
Help Function Descriptions

The following illustration displays the options available on the **Help** subpalette.



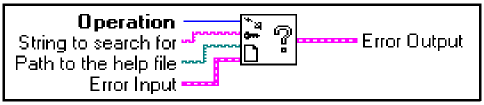
Control Help Window

Modifies the Help window by showing, hiding, or repositioning the window.



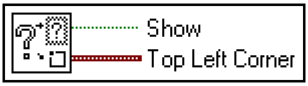
Control Online Help

Controls the online help system by displaying the table of contents of a help file, jumping to a specific point in a help file, or closing the online help system.



Get Help Window Status

Returns the status and the position information for the Help window.

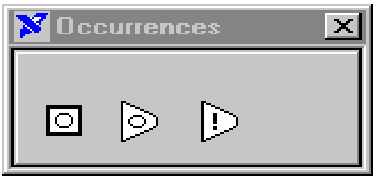


Occurrence Function Descriptions

You can use the occurrence functions to control separate, synchronous activities. In particular, you use these functions when you want one VI or part of a block diagram to wait until another VI or part of a block diagram finishes a task without forcing LabVIEW to poll.

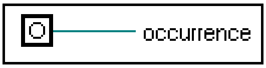
You can perform the same task using global variables, with one loop polling the value of the global until its value changes. However, global variables add overhead, because the loop that waits uses execution time. With occurrences, the polling loop is replaced with a Wait on Occurrence function and does not use processor time. When some diagram sets the occurrence, LabVIEW activates all Wait on Occurrence functions in any block diagrams that are waiting for the specified occurrence.

The following illustration displays the options available on the **Occurrences** subpalette.



Generate Occurrence

Creates an **occurrence** that you can pass to the Wait on Occurrence and Set Occurrence functions.

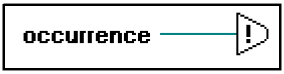


Ordinarily, only one Generate Occurrence node is connected to any set of Wait on Occurrence and Set Occurrence functions. You can connect a Generate Occurrence function to any number of Wait on Occurrence and Set Occurrence functions. You do not have to have the same number of Wait on Occurrence and Set Occurrence functions.

Each Generate Occurrence function on a block diagram represents a single, unique occurrence. In this way, you can think of the Generate Occurrence function as a constant. When a VI is running, every time a Generate Occurrence function executes, the node produces the same value. For example, if you place a Generate Occurrence function inside of a loop, the value produced by Generate Occurrence is the same for every iteration of the loop. If you place a Generate Occurrence function on the block diagram of a reentrant VI, Generate Occurrence produces a different value for each caller.

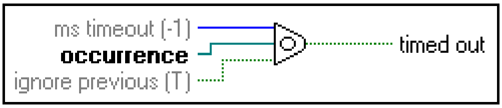
Set Occurrence

Triggers the specified **occurrence**. All block diagrams that are waiting for this occurrence stop waiting.



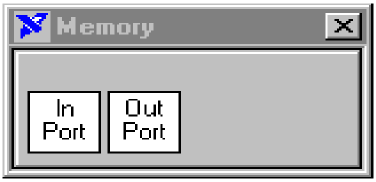
Wait On Occurrence

Waits for the Set Occurrence function to set or trigger the given **occurrence**.



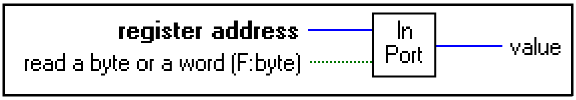
Memory VI Descriptions

The following illustration displays the options available on the **Memory** subpalette.



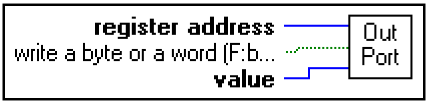
In Port (Windows 3.1 and Windows 95)

Reads a byte or word integer from a specific register address. Because this VI is not available on all platforms, VIs using this subVI are not portable.



Out Port (Windows 3.1 and Windows 95)

Writes a byte or word integer to a specific register address. Because this VI is not available on all platforms, VIs using this subVI are not portable.

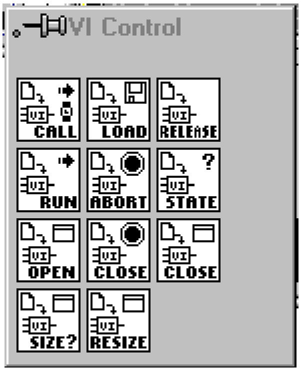


VI Control VI Descriptions

You can use the VI Control VIs to dynamically load, call, and close other VIs. When you call a VI dynamically, you specify whether or not the called VI opens its front panel and then closes the front panel when it finishes executing. You can also pass parameters to and from the dynamically called VI.

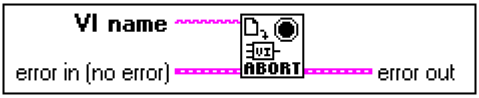
All of these VIs use error cluster inputs and outputs to make error handling easier. If an incoming error is set, the VI does not do anything. The Release Instrument VI, however, releases the specified VI from memory regardless of incoming errors.

The following illustration displays the options available on the **VI Control** subpalette.



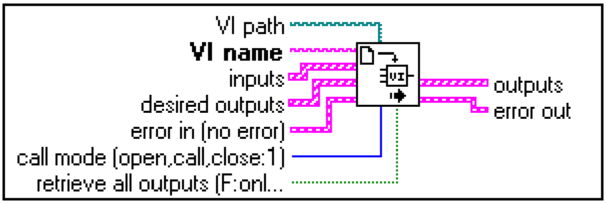
Abort Instrument

Aborts the execution of the specified VI, just as if you clicked the stop button in the specified VI's toolbar.



Call Instrument

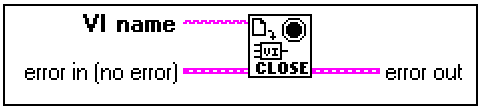
Loads and then calls another VI as long as the VI you are calling is not currently in the VI hierarchy of any running VI, including your main VI. For example, if you have the Serial Port Read VI on your block diagram, you cannot use Call Instrument to call Serial Port Read directly, because it is already in the main VI's hierarchy. However, you can call the Serial Port Read VI if you create a VI that is not part of the main VI's hierarchy. If the called VI has not already been loaded, LabVIEW loads it before the call, and unloads the VI when the call is finished. If you do not want to incur the speed penalty of loading the VI at the time of the call, use the Preload Instrument VI to preload the VI, and then use the Release Instrument VI when you are finished with it. If **error in** contains an error, LabVIEW does not call the VI.



☞ **Note:** ***You can pass data to any control (excluding indicators) on the front panel of the called VI; the controls do not have to be on the connector pane of the called VI.***

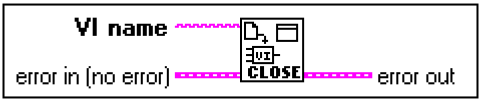
Close Panel

Closes the front panel of a specified VI. If the VI is running it will be aborted.



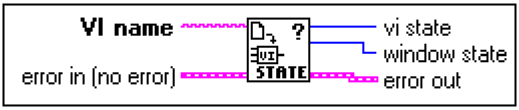
Close Panel No Abort

Closes the front panel of the specified VI. If the VI is running and was loaded using Preload Instrument VI, it will not be aborted. If the VI is running, but it has not been preloaded, it will be aborted.



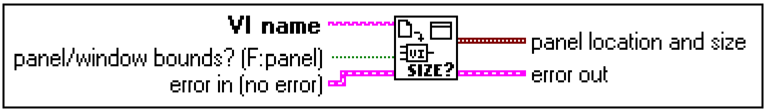
Get Instrument State

Returns the VI execution state (Broken, Idle, or Running) and the panel window state (Closed, Open, or Open and Active). If the VI is not in memory, the error out will be `File Not Found`.



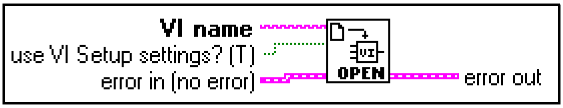
Get Panel Size

Read the size of the panel of a VI that is already in memory. The VI must be in memory but its panel does not need to be open.



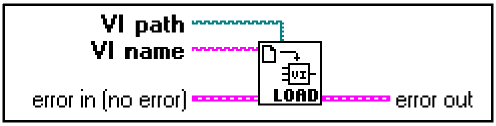
Open Panel

Opens the front panel of the specified VI. The specified VI must already be in memory, either because it was loaded using the Preload Instrument VI, or because it is the subVI of another VI.



Preload Instrument

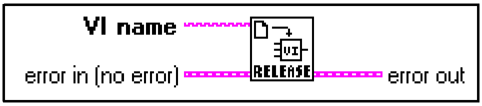
You can use this VI to load another VI into memory. The front panel of the specified VI is not visible when it is loaded. If you want the front panel to be visible, call either Open Panel VI or use the appropriate call mode for the Call Instrument VI.



If you execute a Preload Instrument VI, and it does not return an error, make sure you call the Release Instrument VI when you are finished to remove the loaded VI from memory. If you call the Preload Instrument VI multiple times, you need to balance the calls with Release Instrument VI calls.

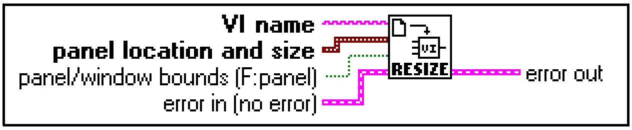
Release Instrument

Use this VI to unload a VI that was loaded using the Preload Instrument VI. If you call Preload Instrument more than once; the specified VI is not unloaded until you call Release Instrument an equal number of times.



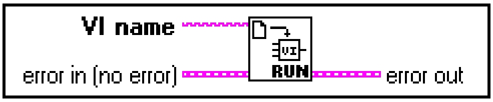
Resize Panel

Resizes and/or moves the front panel of a VI that is already in memory. The VI must be in memory, but its front panel does not have to be open. Consequently, you can size or position a front panel before opening it.



Run Instrument

You can use this VI to run another VI that is in memory with the front panel of the VI in memory open. Run Instrument is different from Call Instrument in that Run Instrument returns immediately after starting the specified VI running, whereas Call Instrument waits for the called VI to complete execution and can pass parameters to and from the called VI. Run Instrument works just as if you selected **Operate»Run**, while Call Instrument functions more like a subVI call.



Introduction to the LabVIEW Data Acquisition VIs

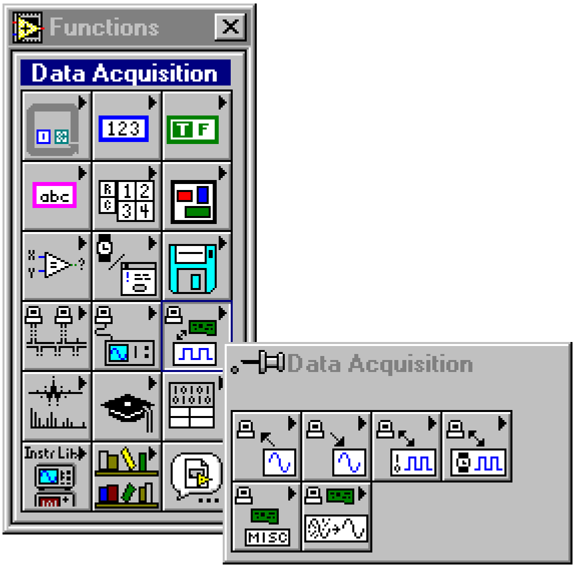
Chapter 13

This chapter contains basic information about the data acquisition (DAQ) VIs and shows where you can find them in LabVIEW. Descriptions of these VIs comprise Chapter 14 through Chapter 29.

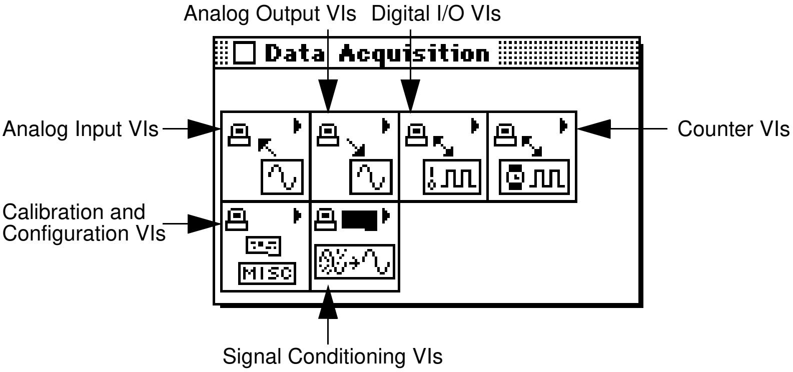
LabVIEW includes a collection of VIs that work with your DAQ hardware devices. With LabVIEW's DAQ VIs you can develop acquisition and control applications.

You can find the DAQ VIs in the **Functions** palette from your block diagram in LabVIEW. The DAQ VIs are located near the bottom of the **Functions** palette.

To access the **Data Acquisition** palette, choose **Functions**»**Data Acquisition**, as shown in the following illustration.



The **Data Acquisition** palette contains six subpalette icons that take you to the different classes of DAQ VIs. The following illustration shows what each of the icons in the **Data Acquisition** palette means.



This section of the manual is organized in the order that the DAQ VI icons appear in the **Data Acquisition** palette from left to right. For example, in this section, the Analog Input VI chapters are followed by the Analog Output VI chapters, which are followed by the Digital I/O VI chapters, and so on. Most often, there are several chapters devoted to one class of DAQ VI in the palette, because many of the VI palettes also contain several subpalettes.

Finding Help Online for the DAQ VIs

You can find helpful information about individual VIs online by using the LabVIEW Help window (**Help**»**Show Help**). When you place the cursor on a VI icon, the wiring diagram and parameter names for that VI appear in the Help window. You can also find information for front panel controls or indicators by placing the cursor over the control or indicator with the Help window open. For more information on the LabVIEW Help window, refer to the *Getting Help* section in Chapter 2, *Creating VIs*, of the *LabVIEW User Manual*.

In addition to the Help window, LabVIEW has more extensive online information available. To access this information, select **Help»Online Reference**. For most block diagram objects, you can select **Online Reference** from the object's pop-up menu to access the online description. You can also access this information by pressing the button shown to the left, which is located at the bottom of LabVIEW's Help window. For information on creating your own online reference files, see the *Creating Your Own Help Files* section in Chapter 25, *Managing Your Applications* of the *LabVIEW User Manual*.

☞ **Note:** ***Use only the inputs that you need on each VI. LabVIEW sets all unwired inputs to their default values. Many of the DAQ function inputs are optional and do not appear in the*** **Simple Diagram Help** ***window. These inputs typically specify rarely-used options. If an input is required, your VI wiring remains "broken" until a value is wired to the input. Required inputs appear in bold in the*** **Help** ***window, recommended inputs appear in plain text, and optional inputs are in gray text. The default values for inputs appear in parentheses beside the input name in the*** **Help** ***window.***

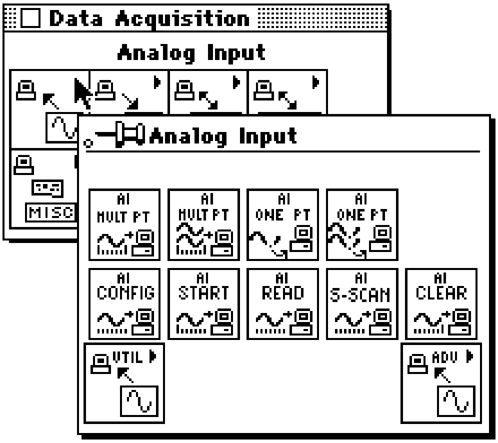
☞ **Note:** ***Some DAQ VIs use an enumerated data type as a control or indicator terminal. If you connect a numeric value to an enumerated indicator, LabVIEW converts the number to the closest enumeration item. If you connect an enumerated control to a number value, the value is the enumeration index. To wire an enumerated control to an enumerated indicator, the enumerated items must match exactly, or you will get a broken wire.***

The Analog Input VIs

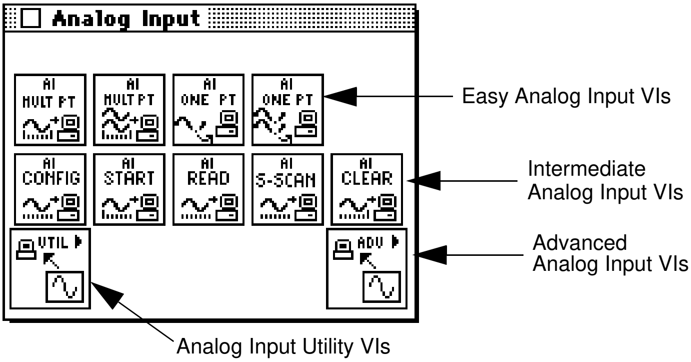
These VIs perform analog input operations.

The Analog Input VIs can be found by choosing **Functions»Data Acquisition»Analog Input**. When you click on the

Analog Input icon in the **Data Acquisition** palette, the **Analog Input** palette pops up, as shown in the following illustration.



There are four classes of Analog Input VIs found in the **Analog Input** palette. The Easy Analog Input VIs, Intermediate Analog Input VIs, Analog Input Utility VIs, and Advanced Analog Input VIs. The following illustrates these VI classes.



Easy Analog Input VIs

Intermediate Analog Input VIs

Advanced Analog Input VIs

Analog Input Utility VIs

Easy Analog Input VIs

The Easy Analog Input VIs perform simple analog input operations. You can run these VIs from the front panel or use them as subVIs in basic applications.

You can use each VI alone to perform a basic analog operation. Unlike intermediate- and advanced-level VIs, Easy Analog Input VIs automatically alert you to errors with a dialog box that asks you to stop the execution of the VI or to ignore the error.

The Easy Analog Input VIs are actually composed of Intermediate Analog Input VIs, which are in turn composed of Advanced Analog Input VIs. The Easy Analog Input VIs provide a basic, convenient interface with only the most commonly used inputs and outputs. For more complex applications, you should use the intermediate- or advanced-level VIs for more functionality and performance.

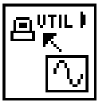
Refer to Chapter 14, *Easy Analog Input VIs*, for specific VI information.

Intermediate Analog Input VIs

You can find intermediate-level Analog Input VIs in two different places in the **Analog Input** palette. You can find the Intermediate Analog Input VIs in the second row of the **Analog Input** palette. The other intermediate-level VIs are in the **Analog Input Utilities** palette, which will be discussed later. The Intermediate Analog Input VIs—AI Config, AI Start, AI Read, AI Single Scan, and AI Clear—are in turn built from the fundamental building block layer, called the Advanced Analog Input VIs. These VIs offer almost as much power as the advanced-level VIs, and they conveniently group the advanced-level VIs into a tidy, logical sequence.

Refer to Chapter 15, *Intermediate Analog Input VIs*, for specific VI information.

Analog Input Utility VIs

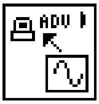


Analog Input Utility Icon

You can access the **Analog Input Utilities** palette by choosing the Analog Input Utility icon from the **Analog Input** palette. The Analog Input Utility VIs—AI Read One Scan, AI Waveform Scan, and AI Continuous Scan—are single-VI solutions to common analog input problems. These VIs are convenient, but they lack flexibility. These three VIs are built from the Intermediate Analog Input VIs in the **Analog Input** palette.

Refer to Chapter 16, *Analog Input Utility VIs*, for specific VI information.

Advanced Analog Input VIs



Advanced Analog Input Icon

You can access the **Advanced Analog Input** palette by choosing the Advanced Analog Input icon from the **Analog Input** palette. These VIs are the interface to the NI-DAQ data acquisition software and are the foundation of the Easy, Utility, and Intermediate Analog Input VIs.

Refer to Chapter 17, *Advanced Analog Input VIs*, for specific VI information.

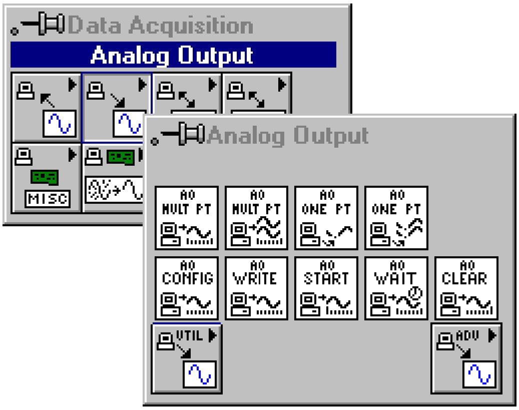
Locating Analog Input VI Examples

For examples of how to use the analog input VIs, see `examples\daq\anlogin\anlogin.llb`

Analog Output VIs

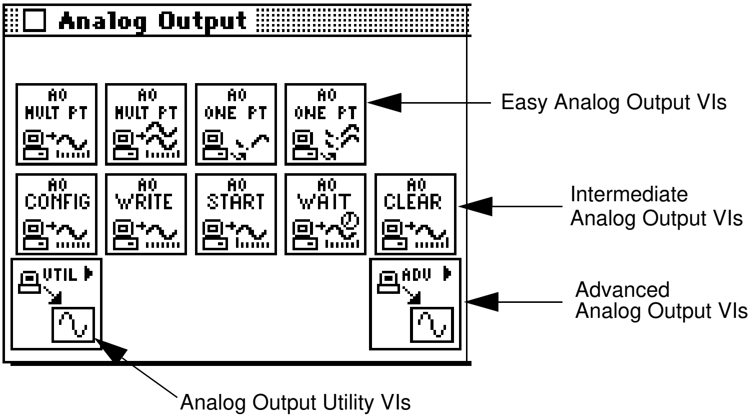
These VIs perform analog output operations.

The Analog Output VIs can be found by choosing **Functions»Data Acquisition»Analog Output**. When you click on the Analog Output icon in the **Data Acquisition** palette, the **Analog Output** palette pops up, as shown in the following illustration.



There are four classes of Analog Output VIs found in the **Analog Output** palette: the Easy Analog Output VIs, Intermediate Analog

Output VIs, Analog Output Utility VIs, and the Advanced Analog Output VIs. The following illustrates these VI classes.



Easy Analog Output VIs

The Easy Analog Output VIs perform simple analog output operations. You can run these VIs from the front panel or use them as subVIs in basic applications.

You can use each VI by itself to perform a basic analog output operation. Unlike intermediate- and advanced-level VIs, Easy Analog Output VIs automatically alert you to errors with a dialog box that asks you to stop the execution of the VI or to ignore the error.

The Easy Analog Output VIs are actually composed of Intermediate Analog Output VIs, which are in turn composed of Advanced Analog Output VIs. The Easy Analog Output VIs provide a basic, convenient interface with only the most commonly used inputs and outputs. For more complex applications, you should use the intermediate- or advanced-level VIs for more functionality and performance.

Refer to Chapter 18, *Easy Analog Output VIs*, for specific VI information.

Intermediate Analog Output VIs

You can find intermediate-level Analog Output VIs in two different places in the **Analog Output** palette. You can find the Intermediate Analog Output VIs in the second row of the **Analog Output** palette. The other intermediate-level VIs are in the **Analog Output Utilities** palette, which will be discussed later. The Intermediate Analog Output VIs—AO Config, AO Write, AO Start, AO Wait, and AO Clear—are in turn built from the fundamental building block layer, called the Advanced Analog Output VIs. These VIs offer almost as much power as the advanced-level VIs, and they conveniently group the advanced-level VIs into a tidy, logical sequence.

Refer to Chapter 19, *Intermediate Analog Output VIs*, for specific VI information.

Analog Output Utility VIs

Analog Output Utility Icon

You can access the **Analog Output Utilities** palette by choosing the Analog Output Utility icon from the **Analog Output** palette. The Analog Output Utility VIs—AI Read One Scan, AI Waveform Scan, and AI Continuous Scan—are single-VI solutions to common analog output problems. These VIs are convenient, but they lack flexibility. These three VIs are built from the Intermediate Analog Output VIs in the **Analog Output** palette.

Refer to Chapter 20, *Analog Output Utility VIs*, for specific VI information.

Advanced Analog Output VIs

Advanced Analog Output Icon

You can access the **Advanced Analog Output** palette by choosing the Advanced Analog Output icon from the **Analog Output** palette. These VIs are the interface to the NI-DAQ software and are the foundation of the Easy, Utility, and Intermediate Analog Output VIs.

Because all these VIs rely on the advanced-level VIs, you can refer to Chapter 21, *Advanced Analog Output VIs*, for additional information on the inputs and outputs and how they work.

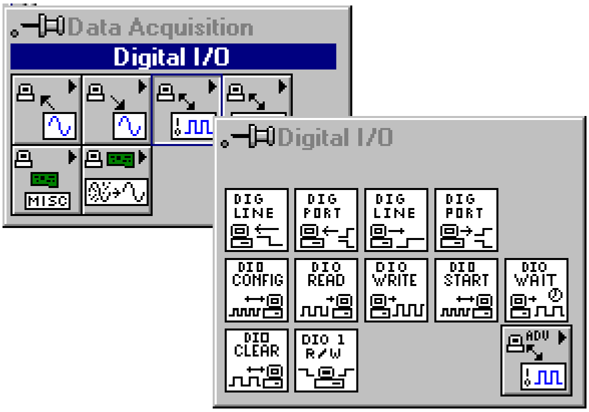
Locating Analog Output VI Examples

For examples of how to use the analog output VIs, see the examples in `examples\daq\anlogout\anlogout.llb`.

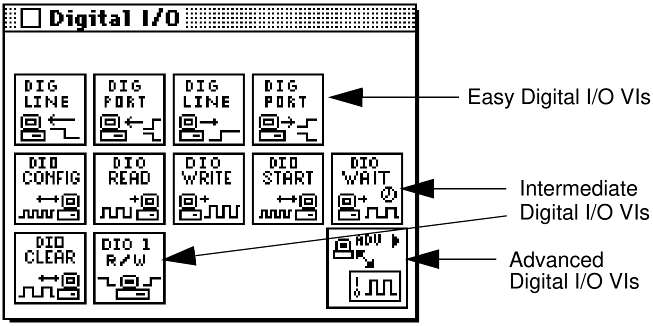
Digital Function VIs

These VIs perform digital operations.

The Digital I/O VIs can be found by choosing **Functions»Data Acquisition»Digital I/O**. When you click on the Digital I/O icon in the **Data Acquisition** palette, the **Digital I/O** palette pops up, as shown in the following illustration.



There are three classes of Digital I/O VIs found in the **Digital I/O** palette. The Easy Digital I/O VIs, Intermediate Digital I/O VIs, and Advanced Digital I/O VIs. The following illustrates these VI classes.



Easy Digital I/O VIs

The Easy Digital I/O VIs perform simple digital operations. You can run these VIs from the front panel or use them as subVIs in basic applications.

You can use each VI by itself to perform a basic digital operation. Unlike intermediate- and advanced-level VIs, Easy Digital I/O VIs automatically alert you to errors with a dialog box that asks you to stop the execution of the VI or to ignore the error.

The Easy Digital I/O VIs are actually composed of Intermediate Digital I/O VIs, which are in turn composed of Advanced Digital I/O VIs. The Easy Digital I/O VIs provide a basic, convenient interface with only the most commonly used inputs and outputs. For more complex applications, you should use the intermediate- or advanced-level VIs for more functionality and performance.

Refer to Chapter 22, *Easy Digital I/O VIs*, for specific VI information.

Intermediate Digital I/O VIs

You can find intermediate-level Digital I/O VIs in the second and third rows of the **Digital I/O** palette. The Intermediate Digital I/O VIs are in turn built from the fundamental building block layer, called the Advanced Digital I/O VIs. These VIs offer almost as much power as the advanced-level VIs, and they conveniently group the advanced-level VIs into a tidy, logical sequence.

Refer to Chapter 23, *Intermediate Digital I/O VIs*, for specific VI information.

Advanced Digital I/O VIs

Advanced Digital I/O Icon

You can access the **Advanced Digital I/O** palette by choosing the Advanced Digital I/O icon from the **Digital I/O** palette. These VIs are the interface to the NI-DAQ software and are the foundation of the Easy, Utility, and Intermediate Digital I/O VIs.

Because all these VIs rely on the advanced-level VIs, you can refer to Chapter 24, *Advanced Digital I/O VIs*, for additional information on the inputs and outputs and how they work.

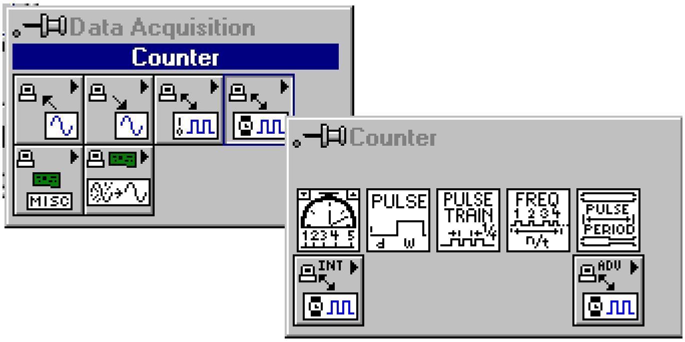
Locating Digital I/O VI Examples

For examples of how to use the Digital I/O VIs, see the examples in `examples\daq\digital\digital.llb`.

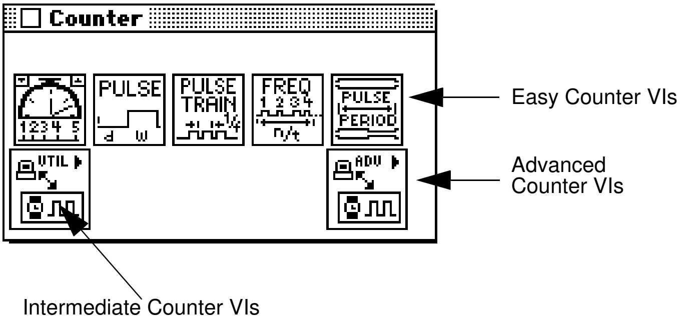
Counter VIs

These VIs perform counting operations.

The Counter VIs can be found by choosing **Functions»Data Acquisition»Counter**. When you click on the Counter icon in the **Data Acquisition** palette, the **Counter** palette pops up, as shown in the following illustration.



There are three classes of Counter VIs found in the **Counter** palette: the Easy, Intermediate, and Advanced Counter VIs. The following illustrates these VI classes.



Easy Counter VIs

The Easy Counter VIs perform simple counting operations. You can run these VIs from the front panel or use them as subVIs in basic applications.

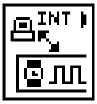
You can use each VI by itself to perform a basic counting operation. Unlike intermediate- and advanced-level VIs, Easy Counter VIs automatically alert you to errors with a dialog box that asks you to stop the execution of the VI or to ignore the error.

The Easy Counter VIs are actually composed of Intermediate Counter VIs, which are in turn composed of Advanced Counter VIs. The Easy Counter VIs provide a basic, convenient interface with only the most commonly used inputs and outputs. For more complex applications, you should use the intermediate- or advanced-level VIs for more functionality and performance.

☞ **Note:** ***An important basic data acquisition concept is to use only the inputs that you need on each VI. Leave the rest of the inputs unwired, and LabVIEW sets them to their default values. In the* Help *window, the most important terminals are labeled in bold, and the least commonly used are in brackets. Values given in parentheses are default values.***

Refer to Chapter 25, *Easy Counter VIs*, for specific VI information.

Intermediate Counter Input VIs

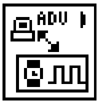


Intermediate Counter VI Icon

You can find the Intermediate Counter VIs in the second row of the **Counter** palette. The Intermediate Counter VIs are in turn built from the fundamental building block layer, called the Advanced Counter VIs. These VIs offer almost as much power as the advanced-level VIs, and they conveniently group the advanced-level VIs into a tidy, logical sequence.

Refer to Chapter 26, *Intermediate Counter VIs*, for specific VI information.

Advanced Counter VIs



You can access the **Advanced Counter** palette by choosing the Advanced Counter icon from the **Counter** palette. These VIs are the interface to the NI-DAQ software and are the foundation of the Easy and Intermediate Counter VIs.

Because all these VIs rely on the advanced-level VIs, you can refer to Chapter 27, *Advanced Counter VIs*, for additional information on the inputs and outputs and how they work.

Locating Counter VI Examples

For examples of how to use the Counter VIs, open the example library by opening `examples\daq\counter\counter.llb`.

Calibration and Configuration VIs

These VIs calibrate specific devices and set and return configuration information.

See Chapter 28, *Calibration and Configuration VIs*, for information on locating these VIs and examples.

Signal Conditioning VIs

These VIs convert analog input voltages read from resistance temperature detectors (RTDs), strain gauges, or thermocouples into units of strain or temperature.

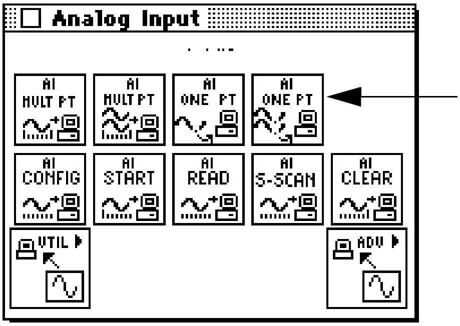
See Chapter 29, *Signal Conditioning VIs*, for information on locating these VIs and examples.

Chapter 14

Easy Analog Input VIs

This chapter describes the Easy Analog Input VIs, which perform simple analog input operations. You can run these VIs from the front panel or use them as subVIs in basic applications.

You can access the Easy Analog Input VIs by choosing **Functions»Data Acquisition»Analog Input**. The Easy Analog Input VIs are the VIs on the top row of the **Analog Input** palette, as shown below.



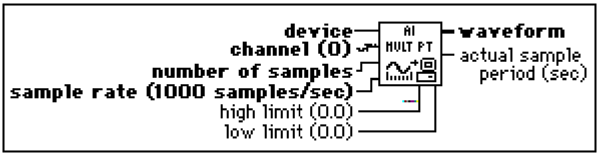
Easy Analog Input VIs

Easy Analog Input VI Descriptions

The following Easy Analog Input VIs are available.

AI Acquire Waveform

Acquires a specified number of samples at a specified sample rate from a single input channel and returns the acquired data.

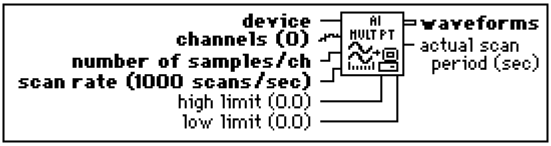


The AI Acquire Waveform VI performs a hardware-timed measurement of a waveform (multiple voltage readings at a specified sampling rate) on a single analog input channel. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel numbers and input limits available with your DAQ device.

AI Acquire Waveforms

Acquires data from the specified channels and samples the channels at the specified scan rate.

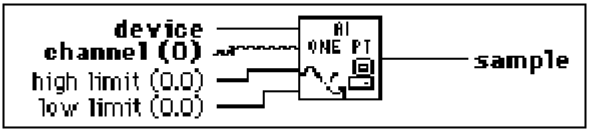


The AI Acquire Waveforms VI performs a timed measurement of multiple waveforms on the specified analog input channels. If an error occurs, a dialog box appear, giving you the option to abort the operation or continue execution.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel numbers and input limits available with your DAQ device.

AI Sample Channel

Measures the signal attached to the specified channel and returns the measured voltage.

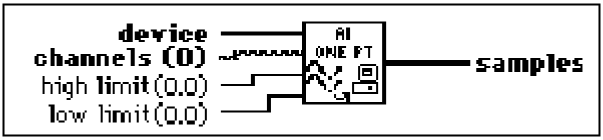


The AI Sample Channel VI performs a single, untimed measurement of a channel. If an error occurs, a dialog box appears giving you the option to stop the VI or continue.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel numbers and input limits available with your DAQ device.

AI Sample Channels

Performs a single voltage reading from each of the specified channels.



The AI Sample Channels VI measures a single voltage from each of the specified analog input channels. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

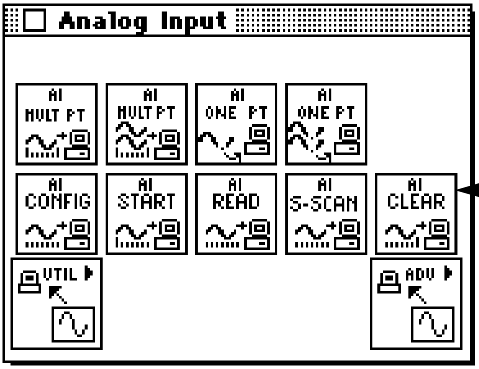
Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel numbers and input limits available with your DAQ device.

Chapter 15

Intermediate Analog Input VIs

This chapter describes the Intermediate Analog Input VIs. These VIs are convenient, but they lack flexibility.

You can access the Intermediate Analog Input VIs by choosing **Functions»Data Acquisition»Analog Input**. The Intermediate Analog Input VIs are the VIs on the second row of the **Analog Input** palette, as shown below.



Intermediate Analog Input VIs

Handling Errors

LabVIEW makes error handling easy with the Intermediate Analog Input VIs. Each intermediate-level VI has an **error in** input cluster and an **error out** output cluster. The clusters contain a Boolean that indicates whether an error occurred, the error code for the error, and the name of the VI that returned the error. If **error in** indicates an error, the VI returns the error information in **error out** and does not continue to run.

☞ **Note:** ***The AI Clear VI is an exception to this rule—this VI always clears the acquisition regardless of whether error in indicates an error.***

When you use any of the Intermediate Analog Input VIs in a While Loop, you should stop the loop if the **status** in the **error out** cluster reads TRUE. If you wire the error cluster to the General Error Handler VI, the VI deciphers the error information and describes the error to you.

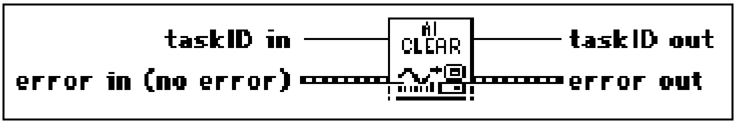
The General Error Handler VI is in **Functions»Time and Dialog** in LabVIEW.

Intermediate Analog Input VI Descriptions

The following Intermediate Analog Input VIs are available.

AI Clear

Clears the analog input task associated with **taskID in**.



The AI Clear VI stops an acquisition associated with **taskID in** and release associated internal resources, including buffers. Before beginning a new acquisition, you must call the AI Config VI. Refer to Chapter 17, *Advanced Analog Input VIs*, for description of the AI Control VI.

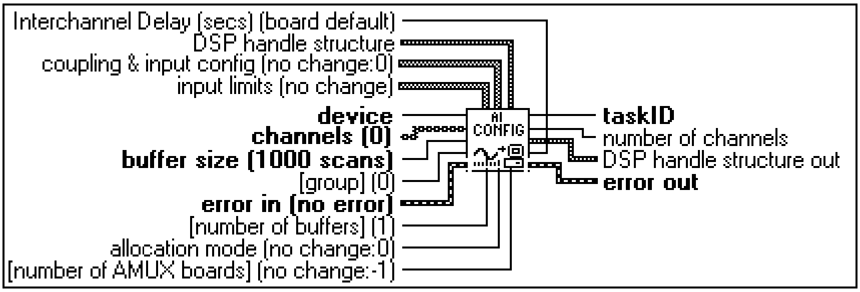
Note: ***The AI Clear VI always clears the acquisition regardless of whether* error in *indicates that an error occurred.***

When you use any of the Intermediate Analog Input VIs in a While Loop, you should stop the loop if the **status** in the **error out** cluster reads TRUE. If you wire the error cluster to the General Error Handler VI, the VI deciphers the error information and describes the error to you.

The General Error Handler VI is in **Functions»Time and Dialog** in LabVIEW. For more information on this VI, refer to your *LabVIEW User Manual*.

AI Config

Configures an analog input operation for a specified set of channels. This VI configures the hardware and allocates a buffer for a buffered analog input operation.



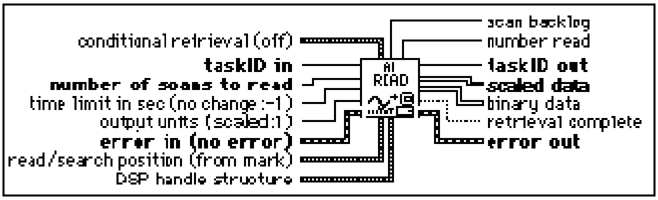
You can allocate more than one buffer only with the following devices.

- **(Macintosh)** NB-A2000, NB-A2100, and NB-A2150
- **(Windows)** EISA-A2000, AT-A2150, and AT-DSP2200

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, input limits, and scanning order you can use with your National Instruments DAQ device.

AI Read

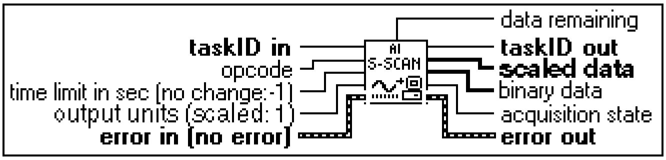
Reads data from a buffered data acquisition.



The AI Read VI calls the AI Buffer Read VI to read data from a buffered analog input acquisition.

AI Single Scan

Returns one scan of data from a previously configured group of channels.



If you have already started an acquisition with the AI Start VI, this VI reads one scan from the acquisition buffer data, or the onboard FIFO if the acquisition is not buffered. If you have not started an acquisition, this VI starts an acquisition, retrieves a scan of data, and then terminates the acquisition. The group configuration determines the channels the VI samples.

If you do not call the AI Start VI, this VI initiates a single scan using the fastest safe channel clock rate. You can alter the channel clock rate with the AI Config VI.

If you run the AI Start VI, a clock signal initiates the scans.

You must use the AI Start VI to set the clock source to external, for externally-clocked conversions.

If clock sources are internal and you do not allocate memory, a timed nonbuffered acquisition begins when you run the AI Start VI. You use this type of acquisition for synchronizing analog inputs and outputs in a point-to-point control application. The following devices do not support timed, nonbuffered acquisitions.

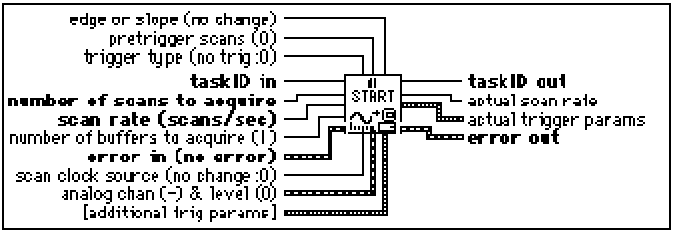
- **(Macintosh)** NB-A2000, NB-A2100, and NB-A2150
- **(Windows)** AT-DSP2200, EISA-A2000, and AT-A2150

☞ **Note:** ***LabVIEW restarts the device in the event of a FIFO overflow during a timed, nonbuffered acquisition.***

When you set **opcode** to `1` for a nonbuffered acquisition, the VI reads one scan from the FIFO and returns the data. If **opcode** is `2`, the VI reads the FIFO until it is empty and returns the last scan read.

AI Start

Starts a buffered analog input operation. This VI sets the scan rate, the number of scans to acquire, and the trigger conditions. The VI then starts an acquisition.



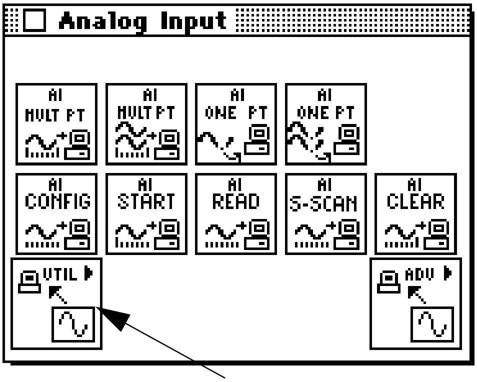
Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, input limits, scanning order, triggers, clocks and you can use with your National Instruments DAQ device.

Chapter 16

Analog Input Utility VIs

This chapter describes the Analog Input Utility VIs. These VIs—AI Read One Scan, AI Waveform Scan, and AI Continuous Scan—are single-VI solutions to common analog input problems. The Analog Input Utility VIs are intermediate-level VIs, so they rely on the advanced-level VIs. You can refer to Chapter 17, *Advanced Analog Input VIs*, for additional information on the inputs and outputs and how they work.

You can access the **Analog Input Utilities** palette by choosing **Functions»Data Acquisition»Analog Input»Analog Input Utilities**. The icon that you must select to access the Analog Input Utility VIs is on the bottom row of the **Analog Input** palette, as shown below.



Analog Input Utility VIs

Handling Errors

LabVIEW makes error handling easy with the intermediate-level Analog Input Utility VIs. Each intermediate-level VI has an **error in** input cluster and an **error out** output cluster. The clusters contain a Boolean that indicates whether an error occurred, the error code for the error, and the name of the VI that returned the error. If **error in** indicates an error, the VI returns the error information in **error out** and does not continue to run.

When you use any of the Analog Input Utility VIs in a While Loop, you should stop the loop if the **status** in the **error out** cluster reads TRUE. If you wire the error cluster to the General Error Handler VI, the VI deciphers the error information and describes the error to you.

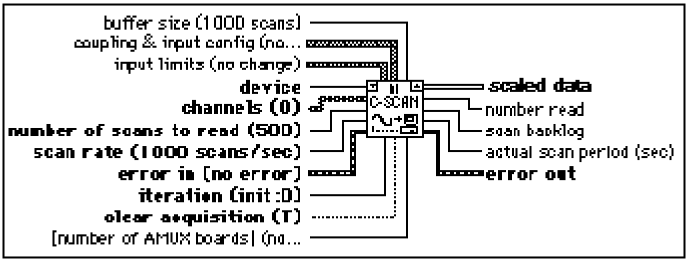
The General Error Handler VI is in **Functions»Time and Dialog** in LabVIEW. For more information on this VI, refer to your *LabVIEW User Manual*.

Analog Input Utility VI Descriptions

The following VIs are available through the Analog Input Utility subpalette.

AI Continuous Scan

Makes continuous, time-sampled measurements of a group of channels, stores the data in a circular buffer, and returns a specified number of scan measurements on each call.



iteration terminal

The AI Continuous Scan VI scans a group of channels indefinitely, as you might do in data logging applications. Place the VI in a While Loop and wire the loop's iteration terminal to the VI **iteration** input.

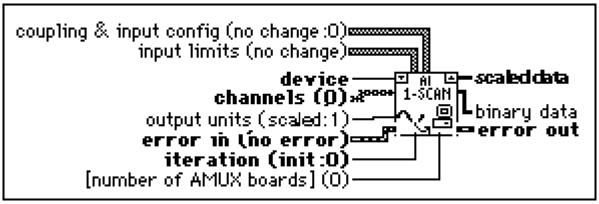
Also wire the condition that terminates the loop to the **clear acquisition** input, inverting the signal if necessary so that it reads TRUE on the last iteration. On iteration 0, the VI calls the AI Config VI to configure the channel group and hardware and allocates a data buffer; the VI calls the AI Start VI to set the scan rate and start the acquisition. On each iteration, the VI calls the AI Read VI to retrieve the number of measurements specified by **number of scans to read**, scales them, and returns the data as an array of voltages. On the last iteration (when **clear acquisition** is TRUE) or if an error occurs, the VI calls the AI Clear VI to clear any acquisition in progress. You should not need to call the AI Continuous Scan VI outside of a loop, but if you do, you can leave the **iteration** and **clear acquisition** inputs unwired.

When calling the AI Continuous Scan VI in a loop to read portions of the data from the ongoing acquisition, you must read the data fast enough so that newly acquired data does not overwrite it. The **scan backlog** output tells you how much data acquired by the VI, but remains unread. If the backlog increases steadily, your new data may eventually overwrite old data. Retrieve data more often, or adjust the **buffer size**, the **scan rate**, or the **number of scans to read** to fix this problem

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, input limits, and scanning order you can use with your National Instruments DAQ device.

AI Read One Scan

Measures the signals on the specified channels and returns the measurements in an array of voltages or binary values.



iteration terminal

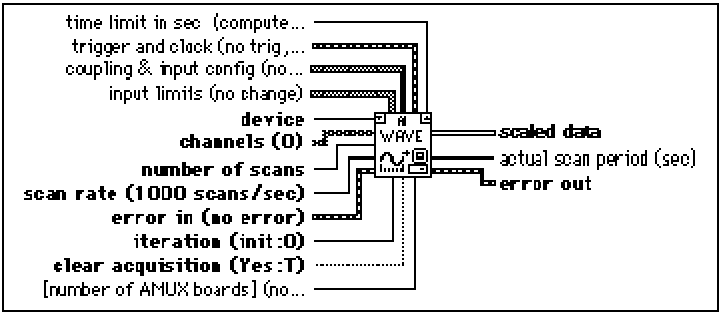
The AI Read One Scan VI performs an immediate measurement of a group of one or more channels. If you place the VI in a loop to take multiple measurements from a group of channels, wire the loop iteration terminal to the VI **iteration** parameter.

On iteration 0, this VI calls the AI Config VI to configure the channel group and hardware, then calls the AI Single Scan VI to measure and report the results. On subsequent iterations, the VI avoids unnecessary configuration and calls only the AI Single Scan VI. If you call the AI Read One Scan VI once to take a single measurement from the group of channels, the **iteration** parameter can remain unwired.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, input limits, and scanning order available with your DAQ device.

AI Waveform Scan

Acquires the specified number of scans at the specified scan rate and returns all the data acquired. You can trigger the acquisition.


time limit in sec (compute...
trigger and clock (no trig,...
coupling & input config (no...
input limits (no change)
device
channels (0)
number of scans
scan rate (1000 scans/sec)
error in (no error)
iteration (init:0)
clear acquisition (Yes:T)
[number of AMUX boards] (no...
AI WAVE
scaled data
actual scan period (sec)
error out

i
iteration terminal

The AI Waveform Scan VI acquires a specified number of scans from a channel group at a specified scan rate. If you place this VI in a loop to take multiple acquisitions from the same group of channels, wire the iteration terminal of the loop to the VI **iteration** input.

Also wire the condition that terminates the loop to the VI **clear acquisition** input, inverting the signal if necessary so that it reads TRUE on the last iteration. On iteration zero, this VI calls the AI Config VI to configure the channel group and hardware and allocate a data buffer. On each iteration, this VI calls the AI Start and AI Read VIs. The AI Start VI sets the scan rate and trigger conditions and starts the acquisition. The VI stores the measurements in the buffer as they are acquired, and the AI Read VI retrieves them from the buffer, scales them, and returns all the data as an array of voltages. On the last iteration (when **clear acquisition** is TRUE) or if an error occurs, the VI also calls the AI Clear VI to clear the acquisition in progress. If you call the AI Waveform Scan VI only once, you can leave **iteration** and **clear acquisition** unwired.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, input limits, scanning order, triggers, and clocks you can use with your National Instruments DAQ device.

☞ **Note:** ***These VIs use an uninitialized shift register as local memory to remember the taskID for the group of channels between VI calls. You normally use one VI in one place on your diagram, but if you use it more than once, the multiple instances of the VI share the same taskID. All calls to one of these VIs configure, read data from, or clear the same acquisition. Occasionally you may want to use each VI in multiple places and have each instance refer to a different taskID (for example, when you measure two devices***

simultaneously). Save a copy of the VI with a new name (for example, AI Waveform Scan R) and make your new VI reentrant.

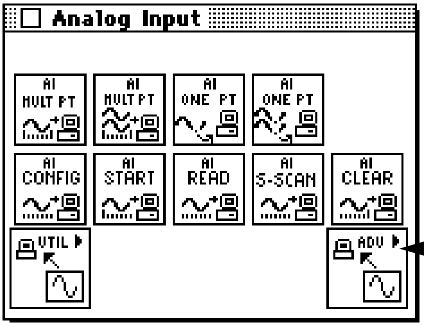
☞ **Note:** ***For all Analog Input Utility VIs, if your program iterates more than 2^{31} - 1 times, do not wire the*** **iteration** ***input to the loop iteration terminal. Instead, set*** **iteration** ***to*** `0` ***on the first loop, then to any positive value on all other iterations. The VI reconfigures and restarts if iteration ≤0.***

Chapter 17

Advanced Analog Input VIs

This chapter contains reference descriptions of the Advanced Analog Input VIs. These VIs are the interface to the NI-DAQ software and are the foundation of the Easy, Utility and Intermediate Analog Input VIs.

You can access the **Advanced Analog Input** palette by choosing **Functions»Data Acquisition»Analog Input»Advanced Analog Input**. The icon that you must select to access the Advanced Analog Input VIs is on the bottom row of the **Analog Input** palette, as shown below.



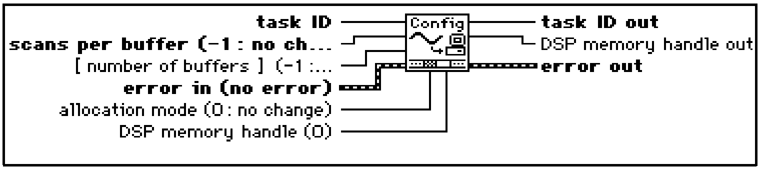
Advanced Analog Input VIs

Advanced Analog Input VI Descriptions

The following Advanced Analog Input VIs are available.

AI Buffer Config

Allocates memory for LabVIEW to store analog input data until the AI Buffer Read VI can deliver it to you. LabVIEW refers to the buffer(s) allocated by the AI Buffer Config VI as internal buffers because you do not have direct access to them.



☞ **Note:** **(Macintosh)** ***If you are using an NB-A2000 with an NB-DMA2800,*** **buffer size** ***and*** **total scans to acquire** ***are both multiples of 32, and your computer has block-mode memory, the driver will automatically use block-mode DMA transfers.***

☞ **Note:** ***When you run the AI Control VI with control code set to*** `4` ***(clear), the VI performs the equivalent of running the AI Buffer Config VI with allocation mode set to*** `1`***. That is, both VIs deallocate the internal analog input data buffers. However, acquisitions that use DSP or expansion card memory are an exception. The AI Control VI does not deallocate DSP memory when clearing an acquisition. You must explicitly call the AI Buffer Config VI to deallocate DSP acquisition buffers.***

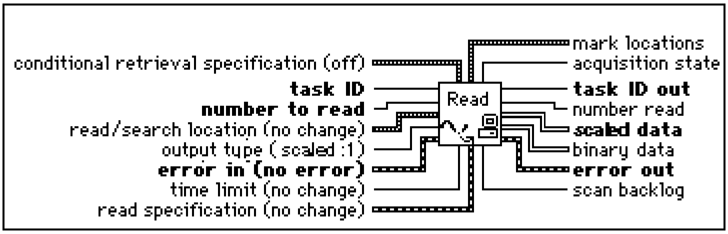
Table 17-1 lists default settings and ranges for the AI Buffer Config VI. The first row gives the values for most devices, and the other rows give the values for devices that are exceptions to the rule.

Table 17-1. AI Buffer Config VI Device-Specific Settings and Ranges

Device	Scans per Buffer		Number of Buffers		Allocation Mode	
	Default Setting	Range	Default Setting	Range	Default Setting	Range
Most devices	100	0, $n \geq 3$	1	0, 1	2	1, 2
Lab-NB Lab-LC	100	$n \geq 3$	1	0, 1	2	1, 2
AT-DSP2200	100	$n \geq 0$	1	$n \geq 0$	2	$1 \leq n \leq 4$
NB-A2000 EISA-A2000 NB-A2100 NB-A2150 AT-A2150	100	$n \geq 0$	1	$n \geq 0$	2	1, 2
5102 devices	100	$n \geq 3$	1	1	2	1, 2

AI Buffer Read

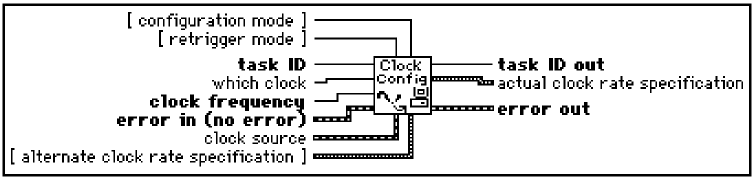
Returns analog input data from the internal data buffer(s).



☞ **Note:** ***When the VI reads from the trigger mark, it does not return data until the acquisition completes for the buffer containing the trigger.***

AI Clock Config

Sets the channel and scan clock rates.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the clocks available with your DAQ device.

For devices that have only a channel clock (Lab-LC, Lab-NB, NB-MIO-16, Lab-PC+, PCI-1200, PC-LPM-16, DAQCard-500, DAQCard-700, and DAQCard-1200), you cannot set independent channel and scan clock rates. Setting one resets the other because the channel rate equals scan rate/number of channels to scan.

For devices that have no channel clock (NB-A2000, NB-A2100, NB-A2150, EISA-A2000, AT-A2150, and AT-DSP2200), setting the channel clock produces an error.

If you specify a value of 0 for the scan clock rate, interval scanning turns off, and channel scanning (or round-robin scanning) proceeds at the channel clock rate. This option is meaningful only for devices with independent channel and scan clocks.

The clock rate is the rate at which LabVIEW samples data or acquires scans. You can express the clock rate three ways—with **clock frequency**, with **clock period**, or with **timebase source**, **timebase signal**, and **timebase divisor**. The VI searches these parameters in that order and sets the clock rate using the first one with a value not equal to `-1`.

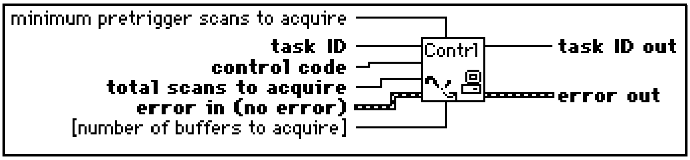
Table 17-2 lists default settings and ranges for the controls of the AI Clock Config VI.

Table 17-2. Device-Specific Settings and Ranges for Controls in the AI Clock Config VI

Device	Configuration Mode		Retrigger Mode	Which Clock		Clock Source	
	Default Setting	**Range**	**Default Setting**	**Default Setting**	**Range**	**Default Setting**	**Range**
AT-MIO-16E-2 AT-MIO-64E-3 NEC-MIO-16E-4 PCI-MIO-16E-1 PCI-MIO-16E-4 PCI-MIO-16XE-10	1	1, 3	no support	1	1, 2	1	1, 2 $4 \le n \le 11$
AT-MIO-16E-10 AT-MIO-16DE-10 AT-MIO-16XE-50 PCI-MIO-16XE-50	1	1, 3	no support	1	1, 2	1	1, 2 $4 \le n \le 9$
AT-A2150 NB-A2150 NB-A2100 NB-A2000 AT-DSP2200 EISA-A2000	1	1, 3	no support	1	1	1	$1 \le n \le 3$
PC-LPM-16 DAQCard-500 DAQCard-700 Lab-PC	1	1, 3	no support	1	1, 2	1	1, 2
Lab-LC Lab-NB NB-MIO-16	1	1, 3	no support	1	2	1	1, 2
All Other Devices	1	1, 3	no support	1	1, 2	1	$1 \le n \le 3$

AI Control

Controls the analog input tasks and specifies the amount of data to acquire.



☞ **Note:** ***You cannot use this VI to start an acquisition when you use a Lab and 1200 Series device, PC-LPM-16, DAQCard-500, or a DAQCard-700 device to scan multiple SCXI channels in multiplexed mode. For this special case, you must use the AI SingleScan VI to acquire data. (For more information about the AI SingleScan VI, refer to its description in this chapter.) However, you can use the AI Control VI for a Lab and 1200 Series device, PC-LPM-16, DAQCard-500, or DAQCard-700 device when you scan SCXI channels in parallel mode or sample a single SCXI channel in multiplexed mode. You can use this VI for an MIO device scanning SCXI channels in either mode.***

☞ **Note:** ***Nonbuffered acquisitions are not supported for the following devices.***

- **(Macintosh)** ***NB-A2000, NB-A2100, or NB-A2150***
- **(Windows)** ***AT-DSP2200, EISA-A2000, or AT-A2150***

Table 17-3 lists default settings and ranges for the AI Control VI.

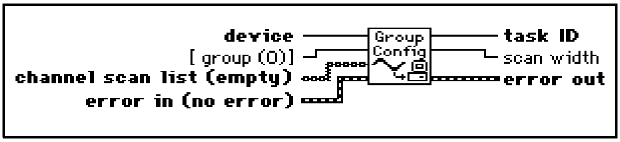
Table 17-3. Device-Specific Settings and Ranges for the AI Control VI

Device	**Control Code**		**Total Scans to Acquire**		**Minimum Pretrigger Scans to Acquire**		**Number of Buffers to Acquire**	
	D S*	**R***	**DS***	**R***	**DS***	**R***	**DS***	**R***
AT-DSP2200, EISA-A2000, AT-A2150, NB-A2000, NB-A2150,	0	0, 1, 4	0	0, *n*≥0	0	0, *n*≥3	1	*n*≥0
PC-LPM-16, DAQCard-500, DAQCard-700	0	0, 1, 4	0	0, *n*≥3	0	no support	1	1
MIO-E Series	0	0, 1, 4	0	0, *n*≥3	0	0, *n*≥3	1	1
5102 Devices	0	0, 1, 4	0	*n*≥0	0	*n*≥0	1	1
All Other Devices	0	0, 1,4	0	0, *n*≥3	0	*n*≥0	1	1

* DS = Default Setting; R = Range

AI Group Config

Defines what channels belong to a group and assigns them.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges and scanning order available with your DAQ device.

Table 17-4 lists default settings and ranges for the AI Group Config VI. The first row of the table gives the values for most devices, and the other rows give the values for devices that are exceptions to the rule.

Table 17-4. Device-Specific Settings and Ranges for the AI Group Config VI

Device	Group		Channel Scan List	
	Default Setting	**Range**	**Default Setting**	**Range**
Most Windows Devices	0	0≤*n*≤15	all channels	0≤*n*≤15
Most Macintosh Devices	0	0≤*n*≤15	all channels	0≤*n*≤15
AT-MIO-64F-5 AT-MIO-64E-3	0	0≤*n*≤15	all channels	0≤*n*≤63
AT-A2150, EISA-A2000	0	0≤*n*≤15	all channels	0≤*n*≤3
AT-DSP2200	0	0≤*n*≤15	all channels	0, 1
Lab-PC+, PCI-1200, DAQCard-1200	0	0≤*n*≤15	all channels	0≤*n*≤7
Lab-LC, Lab-NB	0	0≤*n*≤15	all channels	0≤*n*≤7
NB-A2000, NB-A2150	0	0≤*n*≤15	all channels	0≤*n*≤3
NB-A2100	0	0≤*n*≤15	all channels	0, 1
5102 Devices	0	0≤*n*≤15	all channels	0, 1

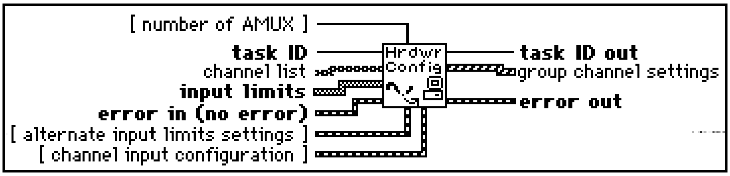
☞ **Note:** ***The Lab-LC, Lab-NB, Lab-PC+, PCI-1200, PC-LPM-16, DAQCard-500, DAQCard-700, and DAQCard-1200 must scan channel lists containing multiple channels from channel n (n ≥ 0) to channel 0 in sequential order, including all channels between n and 0. The NB-A2000, NB-A2150, EISA-A2000, and AT-A2150 allow only the following scan lists: (0), (1), (2), (3), (0, 1), (2, 3), and (0, 1, 2, 3). The NB-A2100 allows the following scan lists: (0), (1), (0, 1), and (1, 0).***

The *channel scan list* ***range shown above is for single-ended mode. Please refer to*** Appendix A, *DAQ Hardware Capabilities*, ***to determine the valid range for channels in differential mode.***

SCXI modules in multiplexed mode must scan channels in ascending consecutive order, starting from any channel on the module. The module order you specify can be arbitrary. SCXI modules in parallel mode must follow the DAQ device restrictions on the order of channel scan lists. Refer to the *Channel, Port, and Counter Addressing* section of Chapter 3, *Basic LabVIEW Data Acquisition Concepts*, in the *LabVIEW Data Acquisition Basics Manual* for information about SCXI channel string syntax.

AI Hardware Config

Configures either the upper and lower input limits or the range, polarity, and gain. The AI Hardware Config VI also configures the coupling, input mode, and number of AMUX-64T devices. The configuration utility determines the default settings for the parameters of this VI.



You can use this VI to retrieve the current settings by wiring **taskID** only or by wiring both **taskID** and **channel list**. If **channel list** is empty, the VI configures channels on a *per group basis*. This means that the configuration applies to all the channels in the group. When you specify one or more channels in **channel list**, the VI configures channels on a *per channel basis*. This means that the configuration applies only to the channels you specify. This VI always returns the current settings for the entire group.

When the configuration is on a per channel basis, **channel list** can contain one or more channels. The channels in **channel list** must belong to the group named by **taskID**. You specify channels the same way you specify them for the AI Group Config VI. If you take multiple samples of a channel within a scan and you want to change the hardware configuration for that channel at each sample, you must supply the settings for each instance of the channel within the scan. If an element of **channel list** specifies more than one channel, the corresponding element of the other arrays applies to all those channels.

The VI applies the values contained in the configuration arrays (**upper input limits, lower input limits**, **coupling**, **range**, **polarity**, **gain**, and **mode**) to the channels in the group (if you configured on a per group basis) or the channels in **channel list** (if you configured on a per channel basis) in the following way. The VI applies the values listed first in the arrays (at index 0) to the first channel in the group or the channel(s) listed in index 0 of **channel list**. The VI applies the values listed second in the configuration arrays (at index 1) to the second channel in the group or channel(s) listed in index 1 of **channel list**. The VI continues to apply the values in this fashion until the arrays are exhausted. If channels in the group or **channel list** remain unconfigured, the VI applies the final values in the arrays to all the remaining unconfigured channels.

Table 17-5 gives examples of this method. The parameter **channel scan list**, which is part of the AI Group Config VI, is used in the following table.

Table 17-5. AI Hardware Config Channel Configuration

Configuration Basis	Array Values	Results
Group	Group **channel scan list** = 1, 3, 4, 5, 7 **channel list** is empty **lower input limit** [0] = –1.0 **upper input limit** [0] = +1.0	All channels in the group have input limits of –1.0 to +1.0.
Group	Group **channel scan list** = 1, 3, 4, 5, 7 **channel list** is empty **lower input limit** [0] = –1.0 **upper input limit** [0] = +1.0 **lower input limit** [1] = 0.0 **upper input limit** [1] = +5.0 **lower input limit** [2] = –10.0 **upper input limit** [2] = +10.0	Channel 1 has input limits of –1.0 to +1.0. Channel 3 has input limits 0.0 to +5.0. Channels 4, 5, and 7 have input limits of –10.0 to +10.0.
Channel	Group **channel scan list** = 1, 3, 4, 5, 7 **channel list** [0] = 1 **channel list** [1] = 3:5 **lower input limit** [0] = –1.0 **upper input limit** [0] = +1.0	Channels 1, 3, 4, and 5 have input limits of –1.0 to +1.0. Channel 7 has the default input limits set by the configuration utility. It is unchanged because it is not listed in **channel list**.

Table 17-5. AI Hardware Config Channel Configuration (Continued)

Configuration Basis	Array Values	Results
Channel	Group **channel scan list** = 1, 3, 4, 5, 7 **channel list** [0] = 1 **channel list** [1] = 3:5 **lower input limit** [0] = –1.0 **upper input limit** [0] = +1.0 **lower input limit** [1] = 0.0 **upper input limit** [1] = +5.0	Channel 1 has input limits of –1.0 to +1.0. Channels 3, 4, and 5 have input limits of 0.0 to +5.0. Channel 7 has the default input limits set by the configuration utility.
Group	Group **channel scan list** = 0, 1, 0, 1 **channel list** is empty **lower input limit** [0] = –1.0 **upper input limit** [0] = +1.0 **lower input limit** [1] = –1.0 **upper input limit** [1] = +1.0 **lower input limit** [2] = –10.0 **upper input limit** [2] = +10.0 **lower input limit** [3] = –10.0 **upper input limit** [3] = +10.0	Channels 0 and 1 have input limits of –1.0 to +1.0 the first time they are sampled and input limits of –10.0 to +10.0 the second time they are sampled.

The **range**, **polarity**, and **gain** determine the lower and upper input limits. When you wire valid **input limit** arrays (that is, arrays of lengths greater than zero) the VI chooses suitable input ranges, polarities, and gains to achieve these **input limits**. The VI ignores the **range**, **polarity**, and **gain** arrays.

If you do not wire the **input limit** arrays, the VI checks **range**, **polarity**, and **gain**. Where the VI finds an array, it sets the corresponding input property to the values in the array. Where the VI does not find an array, it leaves the corresponding input property unchanged.

For some devices and SCXI modules, onboard jumpers set **range**, **polarity**, and/or **gain**. LabVIEW does not alter the settings of jumpered parameters when you specify **input limits**. If LabVIEW cannot achieve the desired **input limits** using the current jumpered settings, it returns a warning.

To override the current jumper values, you must call the AI Hardware Config VI and specify **range**, **polarity**, and/or **gain** explicitly. The configuration utility determines the initial setting for these parameters (the default value is the factory jumper setting).

If a pair of **input limits** values are both `0`, the VI does not change the **input limits**.

SCXI channel hardware configurations are actually a combination of SCXI module and DAQ device settings and require special considerations. The way you specify channels indicates whether LabVIEW alters the SCXI module settings and/or the DAQ device settings. The **input limits** parameter always applies to the entire acquisition path.

When you configure on a per group basis, LabVIEW may alter both SCXI module and DAQ device settings. In this case, **gain** applies to the entire path and is the product of the SCXI channel gain and acquisition device channel gain. LabVIEW sets the highest gain needed on the SCXI module, then adds DAQ device gain if necessary.

When configuration is on a per channel basis, you can specify the channels in one of three ways. The first way is to specify the entire path, as in the following example.

```
OB0!SC1!MD1!CH0:7
```

(Windows) Also, you can specify the path using channel names configured in the DAQ Channel Wizard, as in the following example.

```
temperature
```

If you use either of these methods, LabVIEW can alter both SCXI and DAQ device settings, and **gain** applies to the product of the SCXI channel gain and the DAQ device gain. LabVIEW sets the highest gain needed on the SCXI module, then adds DAQ device gain if necessary.

The second method is to specify the SCXI channel only, as in the following example.

```
SC1!MD1!CH0:7
```

This specification indicates that LabVIEW should alter SCXI settings only. Additionally, **gain** applies only to the SCXI channel.

The third way is to specify the acquisition device channel only, as in the following example.

```
OB0
```

In this case, LabVIEW alters only DAQ device settings. The **gain** parameter applies to the onboard channel only.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, input limits, and scanning order available with your DAQ device.

Tables 17-6 through 17-9 list default settings and ranges for the AI Hardware Config VI. A tilde (~) indicates that the parameter is configurable on a per group basis only. This means you cannot configure it by channel. The first row of these tables give the values for most devices, and the other rows give the values for devices that are exceptions to the rule. If you did not set the default settings with the configuration utility, use the default settings shown in these tables.

Table 17-6. Device-Specific Settings and Ranges for the AI Hardware Config VI

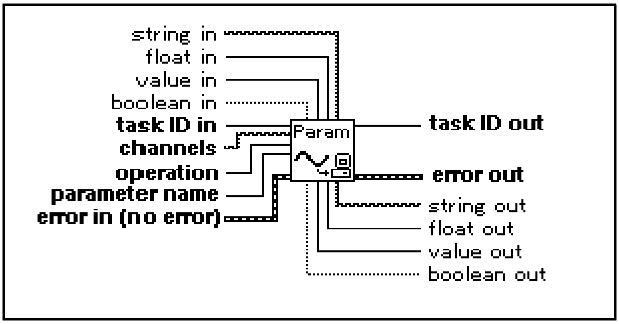
Device	**Channel Input Configuration Cluster**				**Number of AMUX**		**Channel List**
	Coupling		**Input Mode ~**				
	DS*	**R***	**DS***	**R***	**DS***	**R***	**DS***
Most Devices	1	1	1	1 ≤*n*≤3	0	0 ≤*n*≤4	empty
EISA-A2000, NB-A2000	2	1, 2	2	2	0	0	empty
PC-LPM-16, Lab-LC, Lab-NB	1	1	2	2	0	0	empty
Lab and 1200 Series devices	1	1	2	1 ≤*n*≤3	0	0	empty
AT-MIO-16X, AT-MIO-64F-5	1	1	1 (no ~)	1 ≤*n*≤3	0	0 ≤*n*≤4	empty
AT-A2150, AT-DSP-2200, NB-A2100, NB-A2150	1	1, 2	2	2	0	0	empty
DAQCard-500, DAQCard-700	1	1	2	1, 2	0	0	empty
5102 Devices	5	1,2	2	2	0	0	empty

* DS = Default Setting; R = Range

☞ **Note:** ***Channels 0 and 1 and channels 2 and 3 must have the same coupling for the NB-A2150, AT-DSP2200, and AT-A2150.***

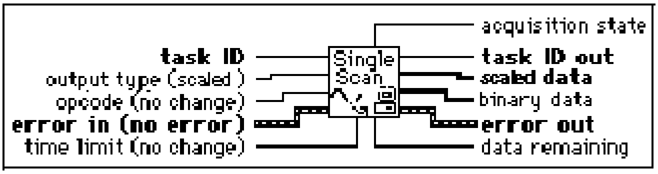
AI Parameter

Configures and retrieves miscellaneous parameters associated with Analog Input of an operation of a device that are not covered with other AI VIs.



AI SingleScan

Returns one scan of data. If you started an acquisition with the AI Control VI, this VI reads one scan of the data from the internal buffer. On the Macintosh and in Windows, the VI reads from the onboard FIFO if the acquisition is nonbuffered. If you have not started an acquisition, this VI starts an acquisition, retrieves a scan of data, and then terminates the acquisition. The group configuration determines the channels the VI sample. This VI does not support 5102 devices.



If you do not call the AI Control VI, this VI initiates a single scan using the fastest and most safe channel clock rate. You can, however, alter the channel clock rate with the AI Clock Config VI.

If you run the AI Control VI with **control code** set to `0` (Start), a clock signal initiates the scans.

If you want externally clocked conversions, you must use the AI Clock Config VI to set the clock source to external.

If clock sources are internal and you do not allocate memory, a timed, nonbuffered acquisition begins when you run the AI Control VI with **control code** set to `0`. This type of acquisition is useful for synchronizing analog inputs and outputs in a point-to-point control application. The following devices do not support timed, nonbuffered acquisitions.

- **(Macintosh)** Lab-NB, Lab-LC, NB-A2000, NB-A2100, and NB-A2150
- **(Windows)** AT-DSP2200, EISA-A2000, and AT-A2150

Note: ***In the event of a FIFO overflow during a timed, nonbuffered acquisition, LabVIEW restarts the device.***

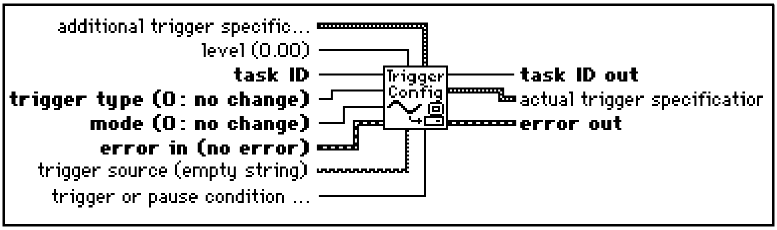
Table 17-7 lists default settings and ranges for the AI SingleScan VI.

Table 17-7. Device-Specific Settings and Ranges for the AI SingleScan VI

Device	Output Type		Opcode		Time Limit	
	DS	R	DS	R	DS	R
AT-DSP2200, EISA-A2000, AT-A2150, NB-A2000, NB-A2100, NB-A2150	1	1≤*n*≤3	1	1	variable	*n*≥0
All Other Devices	1	1≤*n*≤3	1	1≤*n*≤4	1≤*n*≤4	*n*≥0

AI Trigger Config

Configures the trigger conditions for starting the scan and channel clocks and the scan counter.


additional trigger specific...
level (0.00)
task ID
trigger type (0: no change)
mode (0: no change)
error in (no error)
trigger source (empty string)
trigger or pause condition ...
Trigger Config
task ID out
actual trigger specificatior
error out

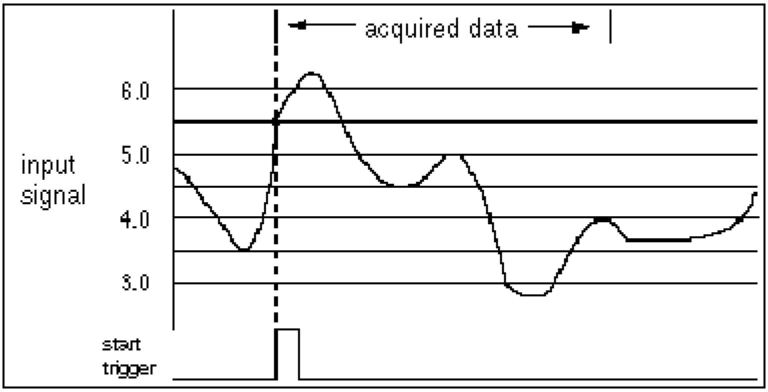
Refer to Appendix A, *DAQ Hardware Capabilities*, for information on the triggers available with your DAQ device. Refer to your E Series device user manual for a detailed description of the triggering capabilities of the device.

The following is a detailed description of trigger types `1` (analog trigger), `2` (digital trigger A), and `3` (digital trigger B) as they apply to three types of applications: posttrigger, pretrigger with software start, and pretrigger with hardware start. The other trigger types are discussed at the end of this section.

Application Type 1: Posttriggered Acquisition (Start Trigger Only)

If **total scans to acquire** is ≥ 0 and **pretrigger scans to acquire** is 0, you are performing a posttriggered acquisition. A **trigger type** of `1` or `2` (analog trigger or digital trigger A, respectively) starts the acquisition (digital trigger B is illegal). You provide a start trigger. Refer to Table 17-10, parts 2 and 3, to determine the default pin to which you connect your trigger signal.On some devices you can specify an alternative source through the **trigger source** parameter.

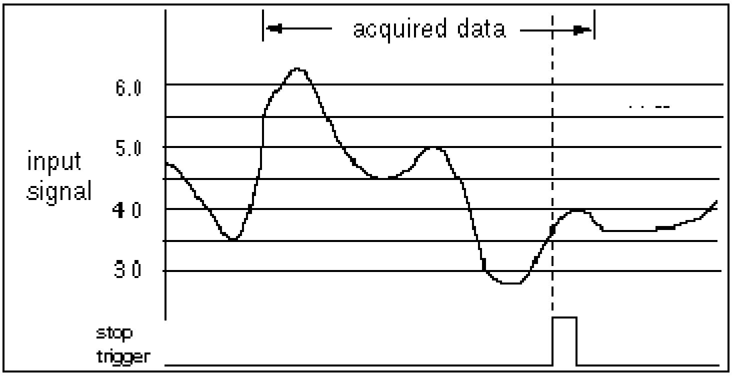
With E Series devices, if you are using an analog trigger and the analog signal is connected to one of the analog input channels, that channel must be first in the scan list. This restriction does not apply if you connect the analog signal to PFI0.



In the above illustration, **total scans to acquire** is `1000` and **pretrigger scans to acquire** is `0`. The start trigger can come from digital trigger A or an analog trigger (**trigger or pause condition** =`1`: Trigger on a rising edge or slope, **level** = `5.5`, **window size** = `0.2`).

Application Type 2: Pretriggered Acquisition (for all trigger types)

If **total scans to acquire** and **pretrigger scans to acquire** are both > 0, a **trigger type** of `1` or `2` (analog trigger or digital trigger A, respectively) starts the acquisition of posttrigger data after the pretrigger data is acquired. The trigger is called a *stop trigger* because the acquisition does not stop until the trigger occurs. A software strobe starts the acquisition. This is a software start pretrigger acquisition. You provide the stop trigger. Refer to Table 17-10, parts 2 and 3, to determine the default pin to which you connect your trigger signal. On some devices, you can specify an alternative source through the **trigger source** parameter.



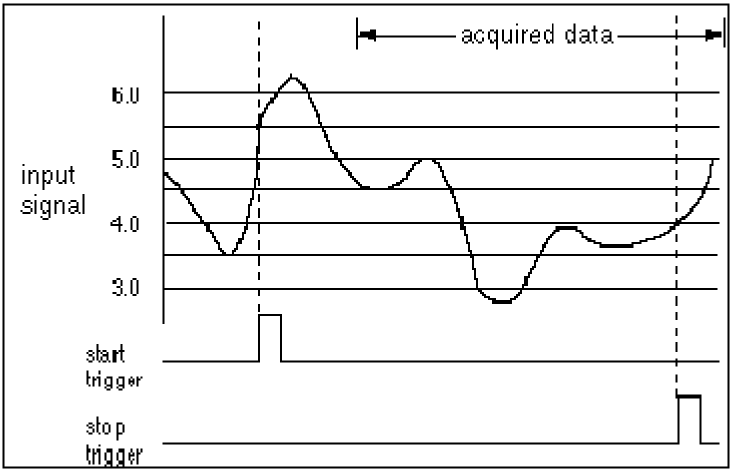
In the above illustration, **total scans to acquire** is `1000` and **pretrigger scans to acquire** is `900`. The stop trigger can come from digital trigger A or an analog trigger (**trigger or pause condition** = `1`: Trigger on rising edge or slope, **level** = `3.7`, **window size** = `0.5`).

With E Series devices, if you are using an analog trigger and the analog signal is connected to an analog input channel, that channel must be the only channel in the scan list (no multiple channel scan allowed). This restriction does not apply if you connect the analog signal to PFI0.

Application Type 3: Pretriggered Acquisition (Start and Stop Trigger)

Application Type 3 is used infrequently. Unless you plan to provide both a start trigger and a stop trigger, skip this section.

On MIO devices, you can enable both the start trigger and the stop trigger. (You must call the AI Trigger Config VI twice to do this.) In this case, a digital or analog trigger signal starts the acquisition rather than a software strobe. This is a hardware start pretriggered acquisition. You provide both the start trigger (as described in *Application Type 1*) and the stop trigger (as described in *Application Type 2*). Refer to Table 17-10 , parts 2 and 3, to determine the default pin to which you connect your trigger signal. On some devices, you can specify an alternative source through the **trigger source** parameter.



In the above illustration, **total scans to acquire** is `1000` and **pretrigger scans to acquire** is `900`. The start trigger can come from digital trigger B or an analog trigger (**trigger or pause condition** = `1:` Trigger on rising edge or slope, **level** = `5.5`, **window size** = `0.2`). The stop trigger can come from digital trigger A or an analog trigger (**trigger or pause condition** = `1:` Trigger on rising edge or slope, **level** = `4.0`, **window size** = `0.2`). Notice that some of the data after the start trigger has been discarded, because all 900 pretrigger scans have been collected and the stop trigger is more than 900 scans away from the start trigger.

When using analog triggering on E Series devices, there are several restrictions that apply, as shown in Table 17-8.

Table 17-8. Restrictions for Analog Triggering on E Series Devices

Start Trigger	Stop Trigger	Restrictions
Digital A	Digital B	None
Digital B	Analog	Analog signal must be connected to PFI0, unless you are scanning only one channel, in which case the input to that channel can be used.
Analog	Digital A	Analog signal must be first in scan list if it is connected to an analog input channel.

A **trigger type** of 4 (digital scan clock gating) enables an external TTL signal to gate the scan clock on and off, effectively pausing and resuming an acquisition.

Channel clock and scan clock are the same on the NB-MIO-16. Therefore, if the scan clock gate becomes FALSE, the current scan does not complete and the scan clock ceases operation. When the scan clock gate becomes TRUE, the scan clock immediately begins operation again, where it left off previously. You wire your signal to the EXTGATE pin.

A **trigger type** of 5 (analog scan clock gating) enables an external analog signal to gate the scan clock on and off, effectively pausing and resuming an acquisition. A trigger type of 6 allows you to use the output of the analog trigger circuitry (ATCOUT) as a general purpose signal. For example, you can use ATCOut to start an analog output operation, or you can count the number of analog triggers appearing at ATCOut.

☞ **Note:** ***Trigger types*** 1, 5, ***and*** 6 ***on E Series devices use the same analog trigger circuitry. All three types can be enabled at the same time, but the last one enabled dictates how the analog trigger circuitry behaves. The E Series restrictions described in the trigger applications apply to all three trigger types.***

Trigger type 5 on E Series devices uses the digital scan clock gate and the analog trigger circuitry. Therefore, enabling trigger type 5 overwrites any settings made for trigger type 4.

For some devices, digital triggering is supported, but for these devices the source is predetermined. Therefore, the **trigger source** parameter is invalid. Table 17-9 shows the pin names on the I/O connector to which you should connect your digital trigger signal.

Table 17-9. Digital Trigger Sources for Devices with Fixed Digital Trigger Sources

Device	Posttriggering	Pretriggering	
	Start Trigger Pin	Start Trigger Pin	Stop Trigger Pin
MIO-16L/H, MIO-16DL/DH	STARTTRIG*	STARTTRIG*	STOPTRIG
NB-MIO-16L/H	STARTTRIG*	no support	no support
AT-MIO-16X, AT-MIO-16F-5, AT-MIO-64F-5	EXTTRIG*	EXTTRIG*	EXTTRIG*
Lab and 1200 Series devices	EXTTRIG	no support	EXTTRIG

Table 17-9. Digital Trigger Sources for Devices with Fixed Digital Trigger Sources (Continued)

Device	Posttriggering	Pretriggering	
	Start Trigger Pin	Start Trigger Pin	Stop Trigger Pin
PC-LPM-16, DAQCard-500, DAQCard-700	no support	no support	no support
AT-DSP2200, EISA-A2000, AT-A2150, NB-A2000, NB-A2100, NB-A2150	EXTTRIG*	no support	EXTTRIG*

* On the AT-MIO-16X, AT-MIO-16F-5, and AT-MIO-64F-5, the same pin is used for both the start trigger and the stop trigger. Refer to your hardware user manual for more details

Table 17-10 lists the default settings and ranges for the AI Trigger Config VI. The first row of each table gives the values for most devices, and the other rows give the values for devices that are exceptions to the rule.

Table 17-10. Device-Specific Settings and Ranges for the AI Trigger Config VI—Part 1

Device	Trigger Type		Mode		Trigger or Pause Condition		Level	
	DS*	R*	DS*	R*	DS*	R*	DS*	R*
Most Devices	2	2, 3	1	1≤*n*≤ 3	no support		no support	
AT-MIO-16E-10, AT-MIO-16DE-10, AT-MIO-16XE-50, PCI-MIO-16XE-50	2	2≤*n*≤4	1	1≤*n*≤ 3	1	1, 2, 7, 8	no support	
AT-MIO-16E-2, AT-MIO-64E-3, NEC-MIO-16E-4	2	1≤*n*≤6	1	1≤*n* ≤3	1	1≤*n*≤8	0	–10 ≤*n*≤ 10
Lab and 1200 Series devices	2	2	1	1≤*n*≤ 3	no support		no support	

Table 17-10. Device-Specific Settings and Ranges for the AI Trigger Config VI—Part 1 (Continued)

Device	Trigger Type		Mode		Trigger or Pause Condition		Level	
	DS*	R*	DS*	R*	DS*	R*	DS*	R*
PC-LPM-16, DAQCard-500, DAQCard-700	no support		no support		no support		no support	
AT-DSP2200, AT-A2150, NB-A2100, NB-A2150	1	1, 2	1	$1 \le n \le 3$	1	1, 2	0	$-2.828 \le n \le 2.828$
EISA-A2000, NB-A2000	1	1, 2	1	$1 \le n \le 3$	1	1, 2	0	$-5.12 \le n \le 5.12$

* DS = Default Setting; R = Range

Table 17-11. Device-Specific Settings and Ranges for the AI Trigger Config VI—Part 2

Device	Trigger Source (Analog)		Additional Trigger Specifications Cluster			
			Window Size		Coupling	
	Default Setting	Range	Default Setting	Range	Default Setting	Range
AT-MIO-16E-2 NEC-MIO-16E-4	0	$0 \le n \le 15$, PFI0	0	$0 \le n \le 20$	no support	
AT-MIO-64E-3	0	$0 \le n \le 63$, PFI0	0	$0 \le n \le 20$	no support	
EISA-A2000, NB-A2000	0	$0 \le n \le 3$	no support		2	1, 2
AT-A2150, NB-A2100, NB-A2150	0	$0 \le n \le 3$	0	$0 \le n \le 5.656$	1	1, 2

Table 17-11. Device-Specific Settings and Ranges for the AI Trigger Config VI—Part 2 (Continued)

Device	Trigger Source (Analog)		Additional Trigger Specifications Cluster			
			Window Size		Coupling	
	Default Setting	Range	Default Setting	Range	Default Setting	Range
AT-DSP2200	0	0, 1	0	0 ≤*n*≤ 5.656	1	1, 2
All Other Devices,	no support		no support		no support	

Device	Trigger Source (Digital)	
	DS	R
E Series Start Trigger	PFI0	PFI 0~9, RTSI 0~6, GPCTR0
E Series Stop Trigger	PFI1	PFI 0~9, RTSI 0~6
E Series Digital Scan Clock Gate	PFI0	PFI 0~9, RTSI 0~6
All Other Devices	no support*	

* See Table 17-9 for devices with fixed digital trigger sources.

Table 17-12. Device-Specific Settings and Ranges for the AI Trigger Config VI—Part 4

Device	Additional Trigger Specifications Cluster					
	Delay		Skip Count		Time Limit	
	DS	R	DS	R	DS	R
EISA-A2000, NB-A2000	0	0≤*n*≤655.35	no support		no support	
AT-A2150	0	0≤*n*≤2.05	no support		no support	

Table 17-12. Device-Specific Settings and Ranges for the AI Trigger Config VI—Part 4 (Continued)

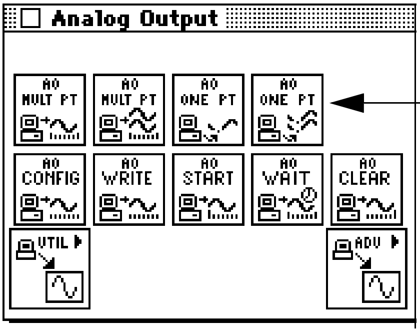
Device	Additional Trigger Specifications Cluster					
	Delay		Skip Count		Time Limit	
	DS	R	DS	R	DS	R
NB-A2100, NB-A2150S	0	0≤n≤32.77	no support		no support	
NB-A2150C	0	0≤n≤16.38	no support		no support	
NB-A2150F	0	0≤n≤17.05	no support		no support	
AT-DSP2200	0	no support	no support		no support	
All Other Devices	no support		no support		no support	

Chapter 18

Easy Analog Output VIs

This chapter describes the Easy Analog Output VIs in LabVIEW, which perform simple analog output operations. You can run these VIs from the front panel or use them as subVIs in basic applications.

You can access the Easy Analog Output VIs by choosing **Functions»Data Acquisition»Analog Output**. The Easy Analog Output VIs are the VIs on the top row of the **Analog Output** palette, as shown below.



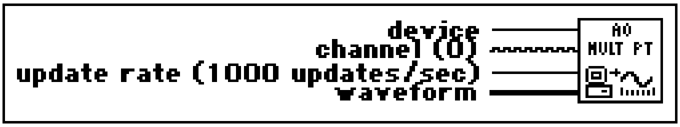
Easy Analog Output VIs

Easy Analog Output VI Descriptions

The following Easy Analog Output VIs are available.

AO Generate Waveform

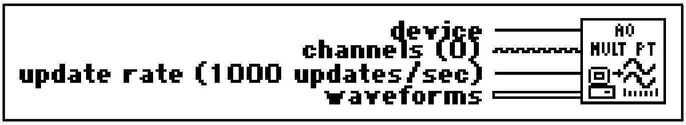
Generates a voltage waveform on an analog output channel at the specified update rate.



The AO Generate Waveform VI generates a multipoint voltage waveform on a specified analog output channel. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

AO Generate Waveforms

Generates multiple voltage waveforms on the specified analog output channels at the specified update rate.

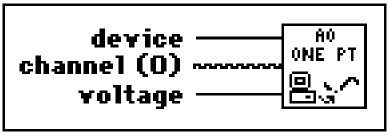


If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel numbers you can use with your DAQ device.

AO Update Channel

Writes a specified voltage value to an analog output channel.

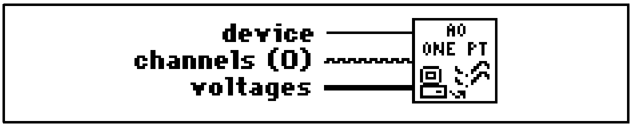


The AO Update Channel VI writes a single update to an analog output channel. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel numbers and output limits available with your DAQ device.

AO Update Channels

Writes voltage values to each of the specified analog output channels.



The AO Update Channels VI updates multiple analog output channels with single voltage values. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

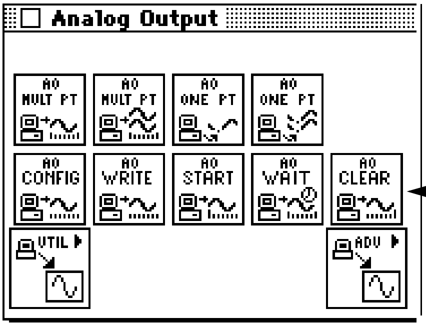
Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel numbers you can use with your DAQ device.

Chapter 19

Intermediate Analog Output VIs

This chapter describes the Intermediate Analog Output VIs. These VIs—AO Write One Update, AO Waveform Gen, and AO Continuous Gen—are single VI solutions to common analog output problems. The intermediate-level VIs are convenient, but they lack flexibility. Because all the VIs in this chapter rely on the advanced layer, you can refer to Chapter 21, *Advanced Analog Output VIs*, for additional information on the inputs and outputs and how they work.

You can access the Intermediate Analog Output VIs by choosing **Functions»Data Acquisition»Analog Output**. The Intermediate Analog Output VIs are the VIs on the second row of the **Analog Output** palette, as shown below.



Intermediate Analog Output VIs

Handling Errors

LabVIEW makes error handling easy with the Intermediate Analog Output VIs. Each intermediate-level VI has an **error in** input cluster and an **error out** output cluster. The clusters contain a Boolean that indicates whether an error occurred, the error code for the error, and the name of the VI that returned the error. If **error in** indicates an error, the VI returns the error information in **error out** and does not continue to run.

☞ **Note:** ***The AO Clear VI is an exception to this rule—this VI always clears the acquisition regardless of whether error in indicates an error.***

When you use any of the Intermediate Analog Output VIs in a While Loop, you should stop the loop if the **status** in the **error out** cluster reads `TRUE`. If you wire the error cluster to the General Error Handler VI, the VI deciphers the error information and describes the error to you.

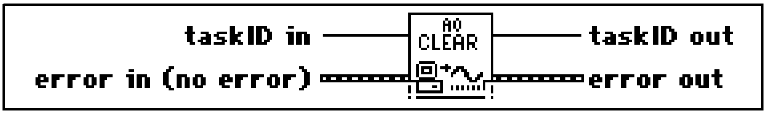
The General Error Handler VI is in **Functions»Time and Dialog** in LabVIEW. For more information on this VI, refer to your *LabVIEW User Manual*.

Analog Output VI Descriptions

The following Analog Output VIs are available.

AO Clear

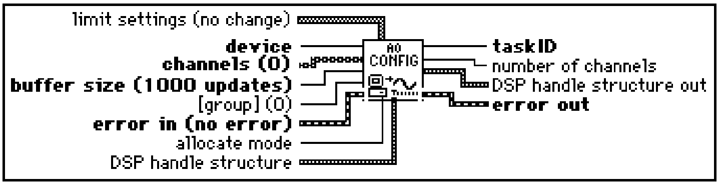
Clears the analog output task associated with **taskID in**.



The AO Clear VI always clears the generation regardless of whether **error in** indicates an error.

AO Config

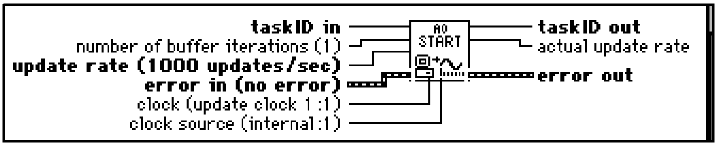
Configures the channel list and output limits, and allocates a buffer for analog output operation.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges and output limits available with your DAQ device.

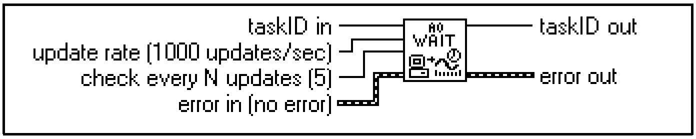
AO Start

Starts a buffered analog output operation. This VI sets the update rate and then starts the generation.



AO Wait

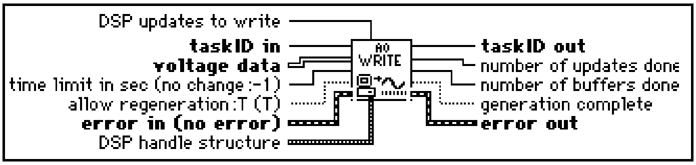
Waits until the waveform generation of the task completes before returning.



Use the AO Wait VI to wait for a buffered, finite waveform generation to finish before calling the AO Clear VI. The AO Wait VI checks the status of the task at regular intervals by calling the AO Write VI and checking its **generation complete** output. The AO Wait VI waits asynchronously between intervals to free the processor for other operations. The VI calculates the wait interval by dividing the **check every N updates** input by the update rate. You should not use the AO Wait VI when you generate data continuously, because the generation never finishes. The AO Clear VI stops a continuous waveform generation.

AO Write

Writes data into the buffer for a buffered analog output operation.

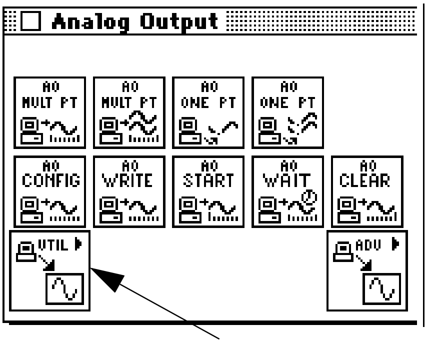


Chapter 20

Analog Output Utility VIs

This chapter describes the Analog Output Utility VIs. The VIs—AO Continuous Generation, AO Waveform Generation, and AO Write One Update—are single-VI solutions to common analog output problems. The Analog Output Utility VIs are intermediate-level VIs, so they rely on the advanced-level VIs. You can refer to Chapter 21, *Advanced Analog Output VIs*, for additional information on the inputs and outputs and how they work.

You can access the **Analog Output Utilities** palette by choosing **Functions»Data Acquisition»Analog Output»Analog Output Utilities**. The icon that you must select to access the Analog Output Utility VIs is on the bottom row of the **Analog Output** palette, as shown below.



Analog Output Utility VIs

Handling Errors

LabVIEW makes error handling easy with the intermediate-level Analog Output Utility VIs. Each intermediate-level VI has an **error in** input cluster and an **error out** output cluster. The clusters contain a Boolean that indicates whether an error occurred, the error code for the error, and the name of the VI that returned the error. If **error in** indicates an error, the VI returns the error information in **error out** and does not continue to run.

When you use any of the Analog Output Utility VIs in a While Loop, you should stop the loop if the **status** in the **error out** cluster reads `TRUE`. If you wire the error cluster to the General Error Handler VI, the VI deciphers the error information and describes the error to you.

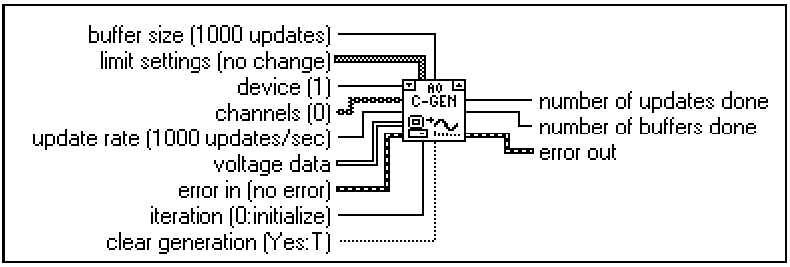
The General Error Handler VI is in **Functions»Utilities** in LabVIEW. For more information on this VI, refer to your *LabVIEW User Manual*.

Analog Output Utility VI Descriptions

The following Analog Output Utility VIs are available.

AO Continuous Gen

Generates a continuous, timed, circular-buffered waveform for the given output channels at the specified update rate. The VI updates the output buffer continuously as it generates the data. If you simply want to generate the same data continuously, use the AO Waveform Gen VI instead.



i
iteration
terminal

You use the AO Continuous Gen VI when your waveform data resides on disk and is too large to hold in memory, or when you must create your waveform in real time. Place the VI in a While Loop and wire the iteration terminal to the VI **iteration** input.

Note: ***If your program iterates more than 2^{31}–1 times, do not wire this VI iteration terminal to the loop iteration terminal. Instead, set* iteration *to* `0` *on the first loop, then to any positive value on all other iterations. The VI reconfigures and restarts if* iteration *≤0.***

Also wire the condition that terminates the loop to the VI's **clear acquisition** input, inverting the signal if necessary so that it is `TRUE` on the last iteration. On iteration 0, the VI calls the AO Config VI to configure the channel group and hardware and to allocate a buffer for the data. It also calls the AO Write VI to write the given voltage data into the buffer, and then the AO Start VI to set the update rate and start the signal generation. On each subsequent iteration, the VI calls the AO Write VI to write the next portion of data into the buffer at the current write position. On the last iteration (when **clear generation** is TRUE) or if an error occurs, the VI also calls the AO Clear VI to clear any generation in progress. Although it is not normally necessary, you can call the AO Continuous Gen VI outside of a loop (that is, to call it only once). But if you do, leave the **iteration** and **clear generation** inputs unwired.

The first call to the AO Write VI sets **allow regeneration** to TRUE, so that the same data can be generated more than once. If you change **allow regeneration** to FALSE, you must write new data fast enough that new data is always available to be generated. If you do not fill the buffer fast enough, you get a regeneration error. To correct this problem, decrease the **update rate**, increase the **buffer size**, increase the amount of data written each time, or write data more often.

(Windows) If you set **allow regeneration** to FALSE, and your device has an analog output FIFO, your **buffer size** must be at least twice as big as your FIFO.

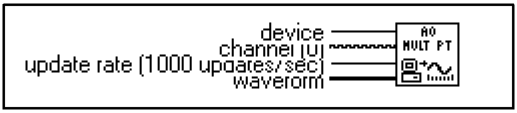
If an error occurs, the VI calls the AO Clear VI to clear any generation in progress, then passes the unmodified error information to **error out**. If an error occurs inside the AO Continuous Gen VI, the AO Clear VI clears any generation in progress and passes its error information out.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges and output limits available with your DAQ device.

☞ **Note:** ***The AO Continuous Gen VI uses an uninitialized shift register as local memory to remember the taskID of the output operation between calls. You normally use this VI in one place on a diagram, but if you use it in more than one place, the multiple instances of the VI share the same taskID. All calls to this VI configure, write data, or clear the same generation. Occasionally, you may want to use this VI in multiple places on the diagram but have each instance refer to a different taskID (for example, when you want to generate waveforms with two devices simultaneously). Save a copy of this VI with a new name (for example, AO Continuous Gen R) and make your new VI reentrant.***

AO Waveform Gen

Generates a timed, simple-buffered or circular-buffered waveform for the given output channels at the specified update rate. Unless you perform indefinite generation, the VI returns control to the LabVIEW diagram only when the generation completes.



iteration terminal

If you place this VI in a loop to generate multiple waveforms with the same group of channels, wire the iteration terminal to the VI **iteration** input.

Note: ***If your program iterates more than 2^{31}–1 times, do not wire this VI iteration terminal to the loop iteration terminal. Instead, set the iteration value to*** `0` ***on the first loop, then to any positive value on all other iterations. The VI reconfigures and restarts if iteration ≤0.***

On iteration 0, the VI calls the AO Config VI to configure the channel group and hardware and to allocate a buffer for the data. On each iteration, the VI calls the AO Write VI to write the data into the buffer, then the AO Start VI to set the update rate and start the generation. If you call the AO Waveform Gen VI only once, you can leave **iteration** unwired. The **iteration** parameter defaults to `0`, which tells the VI to configure the device before starting the waveform generation.

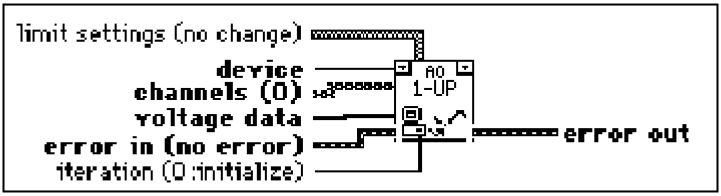
If an error occurs, the VI calls the AO Clear VI to clear any generation in progress, then passes the error information unmodified through **error out**. If an error occurs inside the AO Waveform Gen VI, it clears any generation in progress and passes its error information out.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, output limits, and scanning order available with your DAQ device.

Note: ***The AO Waveform Gen VI uses an uninitialized shift register as local memory to remember the taskID of the output operation between calls. You normally use this VI in one place on your diagram, but if you use it in multiple places, all instances of the VI share the same taskID. All calls to this VI configure, write data, or clear the same generation. Occasionally, you may want to use this VI in multiple places on the diagram, but have each instance refer to a different taskID. Save a copy of this VI with a new name (for example, AO Waveform Gen R) and make the new VI reentrant.***

AO Write One Update

Writes a single voltage value to each of the specified analog output channels.



iteration terminal

The AO Write One Update VI performs an immediate, untimed update of a group of one or more channels. If you place the VI in a loop to write more than one value to the same group of channels, wire the iteration terminal to the VI **iteration** input.

Note: ***If your program iterates more than 2^{31}–1 times, do not wire this VI iteration terminal to the loop iteration terminal. Instead, set the iteration value to*** `0` ***on the first loop, then to any positive value on all other iterations. The VI reconfigures and restarts if iteration ≤0.***

On iteration 0, the VI calls the AO Config VI to configure the channel group and hardware, then calls the AO Single Update VI to write the voltage to the output channels. On future iterations, the VI calls only the AO Single Update VI, avoiding unnecessary configuration. If you call the AO Write One Update VI only once to write a single value to each channel, leave the **iteration** input unwired. Its default value of `0` tells the VI to perform the configuration before writing any data.

Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, output limits, and scanning order available with your DAQ device.

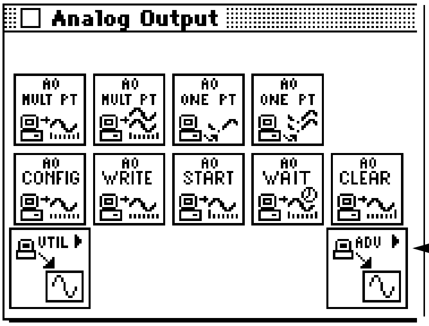
Note: ***The AO Write One Update VI uses an uninitialized shift register as local memory to remember the taskID for the group of channels when calling between VIs. Usually, this VI appears in one place on your diagram. However, if you use it in more than one place, the multiple instances of the VI share the same taskID. All calls to this VI configure or write data to the same group. If you want to use this VI in more than one place on your diagram, and want each instance to refer to a different taskID (for example, to write data with two devices at the same time), you should save a copy of this VI with a new name (for example, AO Write One Update R) and make your new VI reentrant.***

Chapter 21

Advanced Analog Output VIs

This chapter contains reference descriptions of the Advanced Analog Output VIs. These VIs are the interface to the NI-DAQ software and are the foundation of the Easy, Utility and Intermediate Analog Output VIs.

You can access the **Advanced Analog Output** palette by choosing **Functions»Data Acquisition»Analog Output»Advanced Analog Output**. The icon that you must select to access the Advanced Analog Output VIs is on the bottom row of the **Analog Output** palette, as shown below.



Advanced Analog Output VIs

Advanced Analog Output VI Descriptions

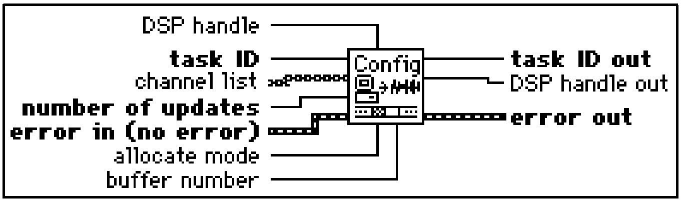
The following Advanced Analog Output VIs are available.

AO Buffer Config

Allocates memory for an analog output buffer. If you are using interrupts, you can allocate a series of analog output buffers and assign them to a group by calling the AO Buffer Config VI multiple times. Each buffer can have its own size. If you are using DMA, you may allocate only one buffer.

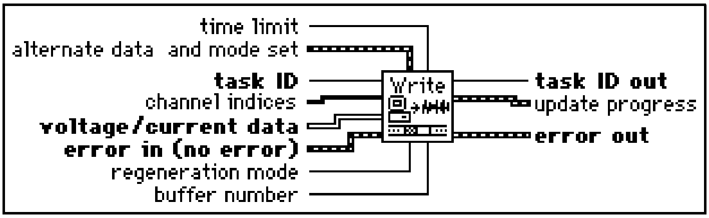
☞ **Note:** **(Macintosh)** ***If you are using the NB-A2100 with the NB-DMA2800, the AO Buffer Write VI restricts the amount of data that can be put into the VI to one-half of the buffer size specified in the AO Buffer Config VI.***

Use the number you assign to the buffer with this VI when you need to refer to this buffer for other VIs.



AO Buffer Write

Writes analog output data to buffers created by the AO Buffer Config VI.

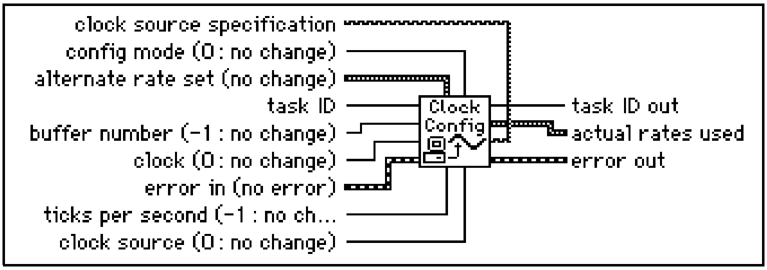


You wire the new data to one of three inputs—**voltage/current data**, **binary data**, or **DSP memory handle**. The VI searches these inputs in that order for the first array with a length greater than zero. The VI then writes the data from this array to the output buffer. The length of the **voltage/curren**t **data** or **binary data** arrays determines the number of updates the VI writes. If **DSP memory handle** points to the source of the data, **updates to write** must indicate how many updates the VI is to write. When no data is wired, this VI is still useful for reporting update progress information.

The total number of updates written to a buffer before you start it can be less than the number of updates you allocated the buffer to hold when you called the AO Buffer Config VI. LabVIEW generates only the updates written to the buffer.

AO Clock Config

Configures an update or interval clock for analog output.

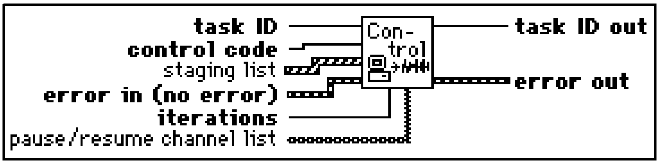

clock source specification
config mode (0: no change)
alternate rate set (no change)
task ID
buffer number (-1 : no change)
clock (0: no change)
error in (no error)
ticks per second (-1 : no ch...
clock source (0: no change)
Clock Config
task ID out
actual rates used
error out

Refer to Appendix A, *DAQ Hardware Capabilities*, for the clocks available with your DAQ device.

You can express clock rates three ways—with **ticks per second**, **seconds per tick**, or the three timebase parameters. The VI searches these parameters in that order and expresses clock rates on the first parameter with a wired valid input. When you configure an update clock, one tick equals one update. When you configure the interval clock, one tick equals one interval.

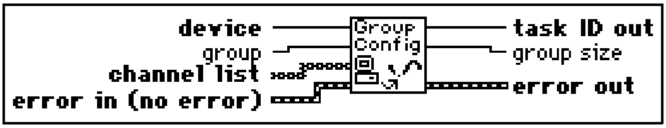
AO Control

Starts, pauses, resumes, and clears analog output tasks.



AO Group Config

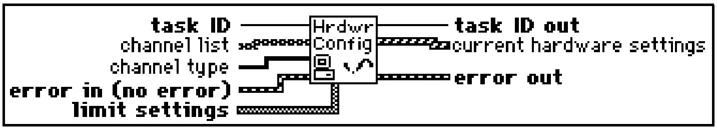
Assigns a list of analog output channels to a group number and produces the taskID that all the other analog output VIs use.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the channels available with your DAQ device.

AO Hardware Config

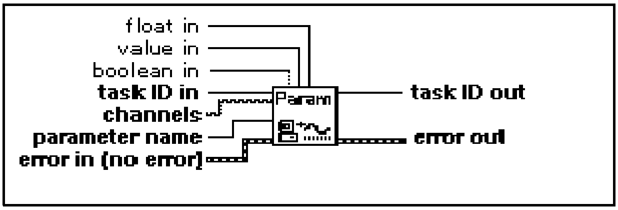
Configures the reference voltage level, output polarity, and the unit of measure for the data of a given channel (volts or milliamperes). This VI always returns the current settings for all the channels in the group.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the channel ranges, and output limits available with your DAQ device.

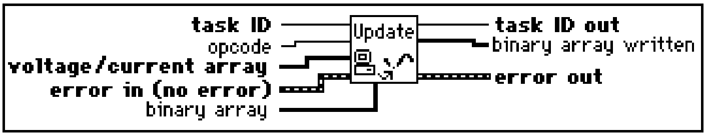
AO Parameter

Sets miscellaneous parameters associated with the Analog Output operation of the devices that are not covered with other Analog Output VIs.



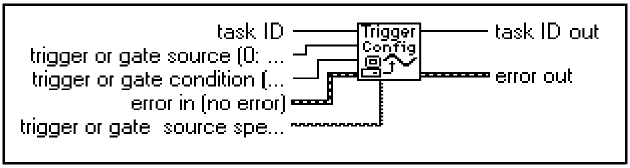
AO Single Update

Performs an immediate update of the channels in the group.



AO Trigger and Gate Config (Windows)

Configures the trigger and gate conditions for analog output operations on E Series devices 5411 devices.

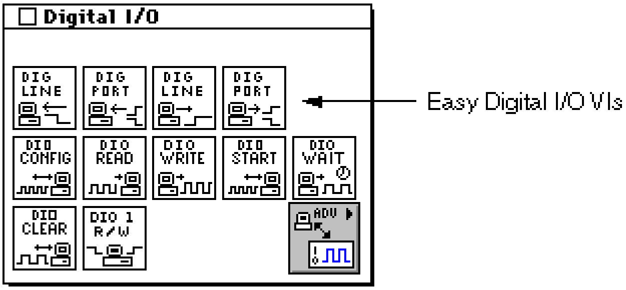

task ID
trigger or gate source (0: ...
trigger or gate condition (...
error in (no error)
trigger or gate source spe...
Trigger
Config
task ID out
error out

Chapter 22

Easy Digital I/O VIs

This chapter describes the Easy Digital I/O VIs, which perform simple digital I/O operations. You can run these VIs from the front panel or use them as subVIs in basic applications.

Access the Easy Digital I/O VIs by choosing **Functions»Data Acquisition»Digital I/O**.



Easy Digital I/O VIs

The Easy Digital I/O VIs are the VIs on the top row of the **Digital I/O** palette. For examples of how to use the Easy Digital I/O VIs, open the example library by opening `examples\daq\digital\digital.llb`.

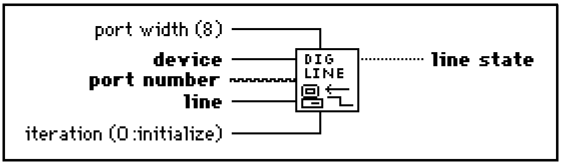
☞ **Note:** ***You must define the high and low limit settings for your board when using the Easy I/O DAQ VIs.***

Easy Digital I/O Descriptions

The following Easy Digital I/Os are available.

Read from Digital Line

Reads the logical state of a digital line on a port that you configure.

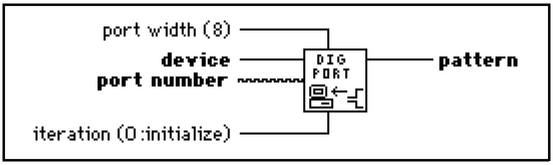


If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

☞ **Note:** ***When you call this VI on a digital I/O port that is part of an 8255 PPI when your iteration terminal is left at 0, the 8255 PPI goes through a configuration phase, where all the ports within the same PPI chip get reset to logic low, regardless of the data direction. The data direction on other ports, however, is maintained. To avoid this effect, connect a value other than 0 to the iteration terminal once you have configured the desired ports.***

Read from Digital Port

Reads a digital port that you configure.

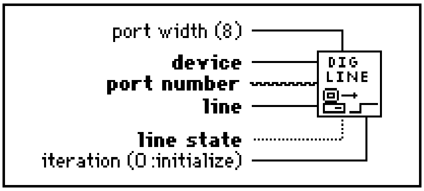


If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

☞ **Note:** ***When you call this VI on a digital I/O port that is part of an 8255 PPI when your iteration terminal is left at 0, the 8255 PPI goes through a configuration phase, where all the ports within the same PPI chip get reset to logic low, regardless of the data direction. To avoid this effect, connect a value other than 0 to the iteration terminal once you have configured the desired ports.***

Write to Digital Line

Sets the output logic state of a digital line to high or low on a digital port that you specify.

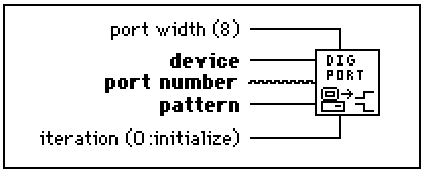


If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

Note: ***When you call this VI on a digital I/O port that is part of an 8255 PPI when your iteration terminal is left at 0, the 8255 PPI goes through a configuration phase, where all the ports within the same PPI chip get reset to logic low, regardless of the data direction. The data direction on other ports, however, is maintained. To avoid this effect, connect a value other than 0 to the iteration terminal once you have configured the desired ports.***

Write to Digital Port

Outputs a decimal pattern to a digital port that you specify.



If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

Note: ***When you call this VI on a digital I/O port that is part of an 8255 PPI when your iteration terminal is left at 0, the 8255 PPI goes through a configuration phase, where all the ports within the same PPI chip get reset to logic low, regardless of the data direction. The data direction on other ports, however, is maintained. To avoid this effect, connect a value other than 0 to the iteration terminal once you have configured the desired ports.***

Chapter 23

Intermediate Digital I/O VIs

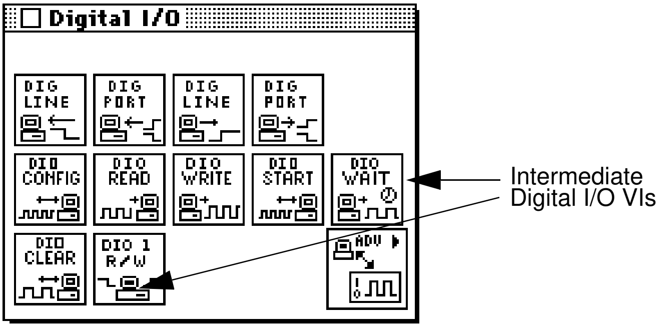
This chapter describes the Intermediate Digital I/O VIs. These VIs are single VI solutions to common digital problems.

For example, the DIO Single Read/Write VI is a single VI solution for non-buffered reads and writes to the ports in your group. The DIO Single Read/Write VI works with any device with digital ports.

You combine the other VIs—DIO Config, DIO Start, DIO Read, DIO Write, DIO Wait, and DIO Clear—to build more demanding applications using buffered digital reads and writes. Your device must support handshaking to use this group of VIs, with the exception of the DIO Single Read/Write VI.

All the VIs described in this chapter are built from the fundamental building block layer, the advanced-level VIs.

You can access the Intermediate Digital I/O VIs by choosing **Functions»Data Acquisition»Digital I/O**. The Intermediate Digital I/O VIs are the VIs on the second and third rows of the **Digital** palette, as shown below.



Handling Errors

LabVIEW makes error handling easy with the Intermediate Digital I/O VIs. Each intermediate-level VI has an **error in** input cluster and an **error out** output cluster. The clusters contain a Boolean that indicates whether an error occurred, the error code for the error, and the name of the VI that returned the error. If **error in** indicates an error, the VI returns the error information in **error out** and does not continue to run.

☞ **Note:** ***The DIO Clear VI is an exception to this rule—this VI always clears the acquisition regardless of whether error in indicates an error.***

When you use any of the Intermediate Digital I/O VIs in a While Loop, you should stop the loop if the **status** in the **error out** cluster reads `TRUE`. If you wire the error cluster to the General Error Handler VI, the VI deciphers the error information and describes the error to you.

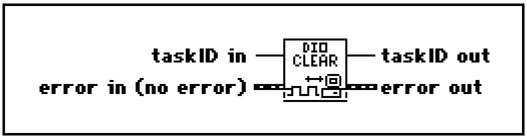
The General Error Handler VI is in **Functions»Time and Function** in LabVIEW. For more information on this VI, refer to your *LabVIEW User Manual*.

Intermediate Digital I/O VI Descriptions

The following Intermediate Digital I/O VIs are available.

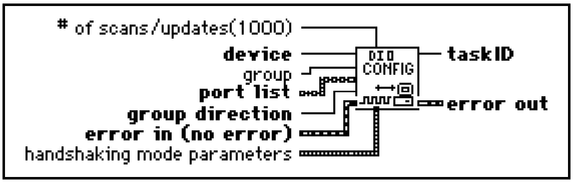
DIO Clear

Calls the Digital Group Buffer Control VI to halt a transfer and clear the group.



DIO Config

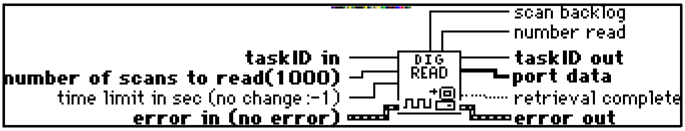
The DIO Config VI calls the advanced Digital Group Config VI to assign a list of ports to the group, establish the group's direction, and produce the **taskID**. The VI then calls the Digital Mode Config VI to establish the handshake parameters, which only affect the operation of the DIO-32 devices. Finally, the VI calls the Digital Buffer Config VI to allocate a buffer to hold the scans as they are read or the updates to be written.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the ports and directions available with your DAQ device.

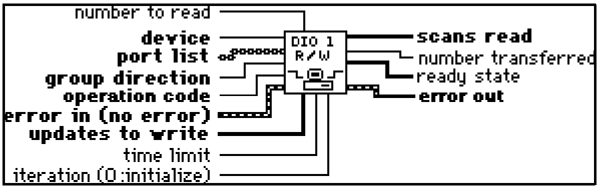
DIO Read

Calls the Digital Buffer Read VI to read data from the internal transfer buffer and returns the data read in **pattern**.



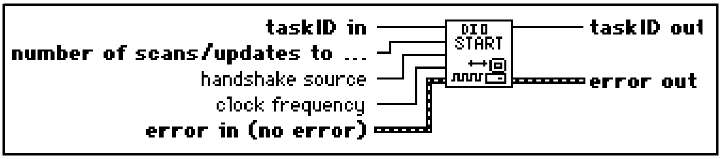
DIO Single Read/Write

Reads or writes digital data to the ports specified in the port list. This single VI configures and transfers data. When you use this VI in a loop, wire the **iteration** counter to the **iteration** input so that port configuration takes place only once.



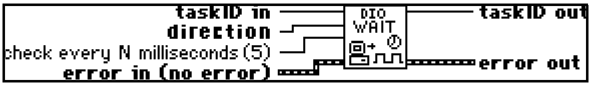
DIO Start

Starts a buffered digital I/O operation. This VI calls the Digital Clock Config VI to set the clock rate if the internal clock produces the handshake signals, and then starts the data transfer by calling the Digital Buffer Control VI.



DIO Wait

Waits until the digital buffered input or output operation completes before returning. For input, the VI detects completion when the acquisition state returned by the Digital Buffer Read VI finishes with or without backlog. For output, the VI detects completion when the **generation complete** indicator of the DIO Write VI is TRUE.

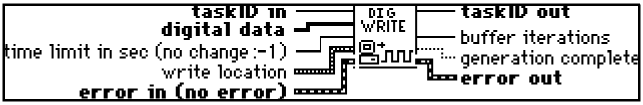


Refer to Appendix A, *DAQ Hardware Capabilities*, for the handshake modes available with your DAQ device.

DIO Write

Calls the Digital Buffer Write VI to write to the internal transfer buffer.

(Macintosh) You must fill the buffer with data before you use the DIO Start VI to begin the digital output operation. You can call the DIO Write VI after the transfer begins to retrieve status information.

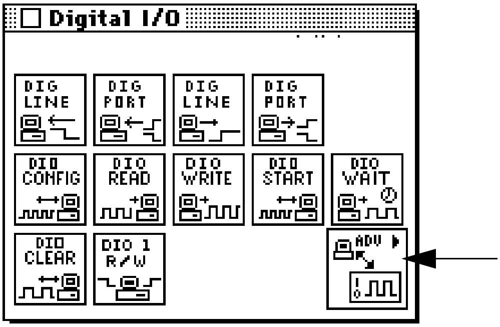


Chapter 24

Advanced Digital I/O VIs

This chapter describes the Advanced Digital I/O VIs, which include the digital port and digital group VIs. You use the digital port VIs for immediate reads and writes to digital lines and ports. You use the digital group VIs for immediate, handshaked, or clocked I/O for multiple ports. These VIs are the interface to the NI-DAQ software and the foundation of the Easy and Intermediate Digital I/O VIs.

You can access the **Advanced Digital I/O** palette by choosing **Functions»Data Acquisition»Digital I/O»Advanced Digital I/O**. The icon that you must select to access the Advanced Digital I/O VIs is on the bottom row of the **Digital I/O** palette, as shown below.



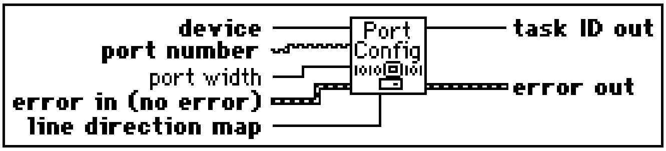
Advanced Digital I/O VIs

Digital Port VI Descriptions

The digital port VIs perform immediate digital reads and writes only.

DIO Port Config

Establishes a port configuration. You can use the **taskID** that this VI returns only in digital port VIs.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the ports and directions available with your DAQ device.

Table 24-1 shows the physical port widths you can use.

Table 24-1. Physical Port Widths of Digital Ports

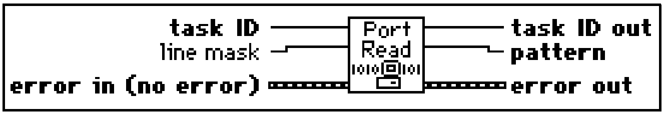
Device	Ports	Physical Port Width
MIO-16L/H	0, 1	4 bits
AT-MIO-16D	0, 1 2, 3, 4	4 bits 8 bits
Most E Series Devices	0	8 bits
AT-MIO-10DE-10	0, 2, 3, 4	8 bits
AT-AO-6/10	0, 1	4 bits
PC-TIO-10, NB-TIO-10, AO-2DC Devices	0, 1	8 bits
PC-LPM-16, PC-LPM-16PnP, DAQCard-700	0, 1	8 bits (cannot be combined)
DAQCard-500, 516 Devices	0, 1	4 bits
Lab and 1200 Series Devices, DIO-24 Devices	0, 1, 2	8 bits
DIO-96 Devices	0 through 11	8 bits
AT-DIO-32F, NB-DIO-32F	0 through 3	8 bits
	4	3 bits (cannot be combined)

Table 24-1. Physical Port Widths of Digital Ports (Continued)

Device	Ports	Physical Port Width
DIO32HS		
SCXI-1160	0	16 bits
SCXI-1161	0	8 bits
SCXI-1162, SCXI-1162HV, SCXI-1163, SCXI-1163R	0	32 bits

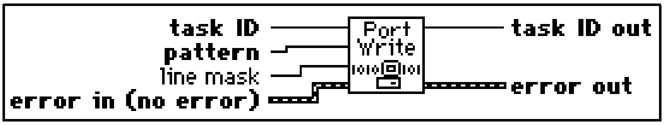
DIO Port Read

Reads the port identified by **taskID** and returns the pattern read in **pattern**.



DIO Port Write

Writes the value in **pattern** to the port identified by **taskID**.

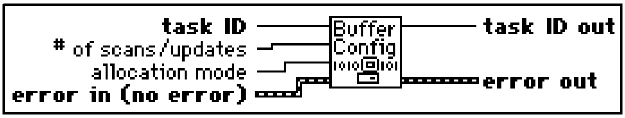


Digital Group VI Descriptions

The digital group VIs perform immediate, handshaked, or clocked digital I/O.

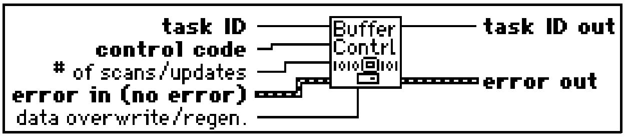
Digital Buffer Config

Allocates memory for a digital input or output buffer.



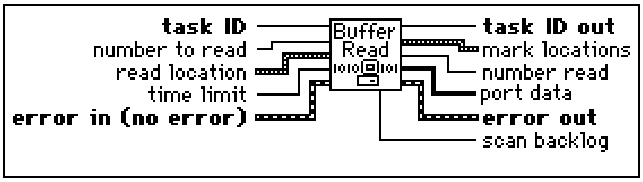
Digital Buffer Control

Starts an input or output operation.



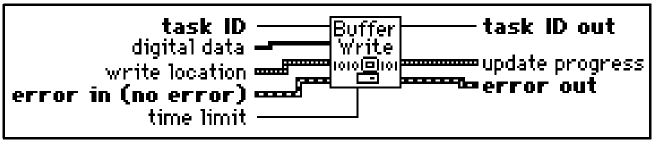
Digital Buffer Read

Returns digital input data from the internal data buffer.



Digital Buffer Write

Writes digital output data to the buffer created by the Digital Buffer Config VI. The write always begins at the write mark. After a write, the write mark points to the update following the last update written.

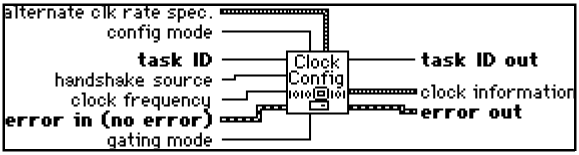


(Macintosh) Fill the buffer with data before you use the Digital Buffer Control VI to begin the digital output operation. You can call the Digital Buffer Write VI after the transfer begins to retrieve status information.

The total number of updates written to a buffer before you start it can be less than the number of updates you allocated the buffer to hold when you called the Digital Buffer Config VI. The VI generates only the updates written to the buffer.

Digital Clock Config

Configures a DIO-32 device to produce handshake signals based on the output of a clock for timed digital I/O.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the clocks available with your DAQ device.

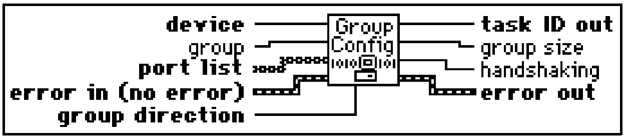
The following example illustrates how to use the three timebase parameters to specify a clock rate. Assume these parameters have the following settings:

timebase source:	1
timebase signal:	1,000,000.0 Hz
timebase divisor:	25

In this case, the ticks per second rate is 1,000,000.0 divided by 25, so LabVIEW updates the digital group 40,000 times per second.

Digital Group Config

Defines a digital input or output group. You can use the **taskID** this VI returns only in the digital group VIs.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the ports and directions available with your DAQ device.

☞ **Note:** ***The same port cannot belong to two different groups. If you configure a group to use a specified port, that port must be one that is not already defined in another group or you will get an error.***

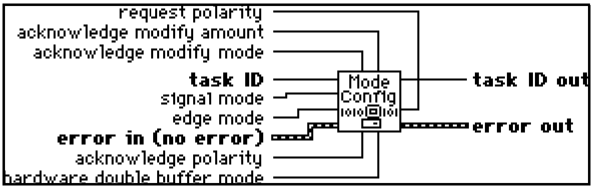
MIO devices (except for the AT-MIO-16D and the AT-MIO-16DE-10), as well as the NB-TIO-10, LPM devices, DAQCard-500, 516 devices, DAQCard-700, PC-TIO-10, AO-2DC devices, PC-OPDIO-16, and AT-AO-6/10, do not allow handshaking. The digital port VIs are more appropriate for these devices. The AT-MIO-16D and

AT-MIO-16DE-10 do not allow handshaking if **port list** includes ports 0, 1, and/or 4. The DIO-96 devices do not allow handshaking if **port list** includes ports 2, 5, 8, and/or 11. The DIO-24 and Lab and 1200 Series devices do not allow handshaking if **port list** includes port 2. The DIO-32F allows handshaking for the following configurations only:

- A group containing any one port
- A group containing ports 0 and 1, or ports 2 and 3, in that order
- A group containing ports 0, 1, 2, and 3, in that order

Digital Mode Config

Configures the handshaking characteristics for DIO-32 devices.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the handshake modes available with your DAQ device.

DIO Parameter

Configures and retrieves miscellaneous parameters associated with digital input and output that are not configured by other DIO VIs.

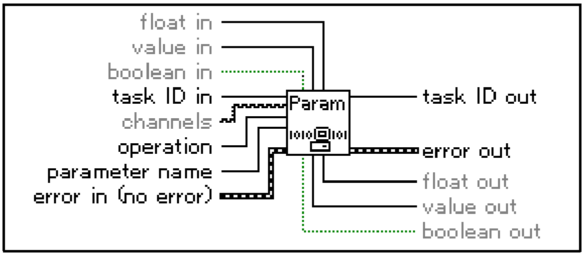


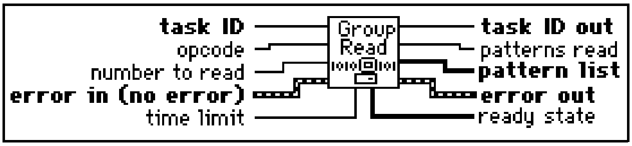
Table 24-2 lists device specific parameters and legal ranges for devices.

Table 24-2. Device specific parameters and legal ranges for devices

Device	Parameter Name	Support	Setting Possible	Input/output you should use	Legal Values	Default Value
VXI-DIO-128	0: Input Port Logic Threshold	per input port	yes	channels, float in, float out	N/A	N/A

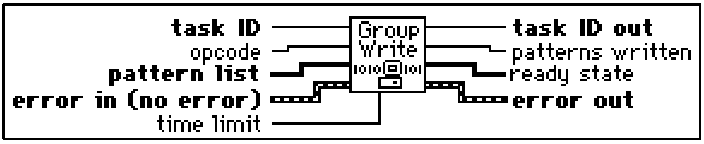
Digital Single Read

Reads the ports that belong to the group identified by **taskID** and returns the patterns read.



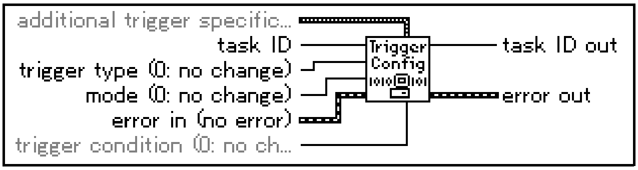
Digital Single Write

Writes the data in **pattern array** to the ports that belong to the group identified by **taskID**.



Digital Trigger Config

Configures the trigger condition for starting and/or stopping a digital pattern generation operation. This VI is only valid when the Digital Clock Config VI has its **handshake source** parameter set to 1 or 4 (internal or external pattern generation w/ external clock).

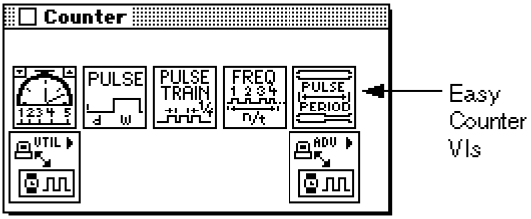

additional trigger specific...
task ID
trigger type (0: no change)
mode (0: no change)
error in (no error)
trigger condition (0: no ch...
Trigger Config
task ID out
error out

Chapter 25

Easy Counter VIs

The Easy Counter VIs perform simple counting operations. You can run these VIs from the front panel or use them as subVIs in basic applications.

You can access the Easy Counter VIs by choosing **Functions»Data Acquisition»Counter**. The Easy Counter VIs are the VIs on the top row of the **Counter** palette.



This chapter describes the high-level VIs for programming counters on the MIO, TIO, and other devices with the Am9513 or DAQ-STC counter/timer chips. These VIs call the Intermediate Counter VIs to generate a single delayed TTL pulse, a finite or continuous train of pulses, and to measure the frequency, pulse width, or period of a TTL signal.

☞ **Note:** ***These VIs do not work with Lab and 1200 Series devices, DAQCards, and other devices that have the 8253 chip. Use the intermediate-level ICTR Control for those devices. Refer to Chapter 26, Intermediate Counter VIs for more information on the ICTR Control VI.***

Some of these VIs use other counters in addition to the one specified. In this case, a logically adjacent counter is chosen, which is referred to as counter+1 when it is the adjacent, logically higher counter and counter–1 when it is the adjacent, logically lower counter.

For a device with the Am9513 chip, if the counter is 1, then counter+1 is counter 2 and counter–1 is counter 5.

See the Adjacent Counters VI described in Chapter 26, *Intermediate Counter VIs*, for more information.

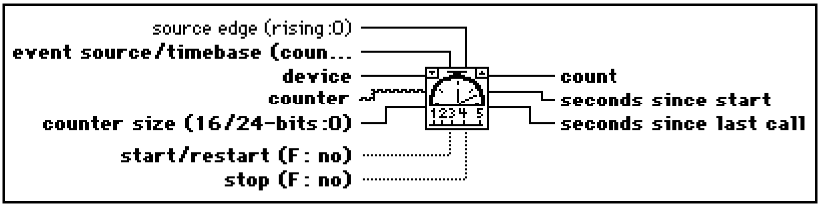
For examples of how to use the Easy Counter VIs, open the example library by opening `examples\daq\counter\counter.llb`.

Easy Counter VI Descriptions

The following Easy Counter VIs are available.

Count Events or Time

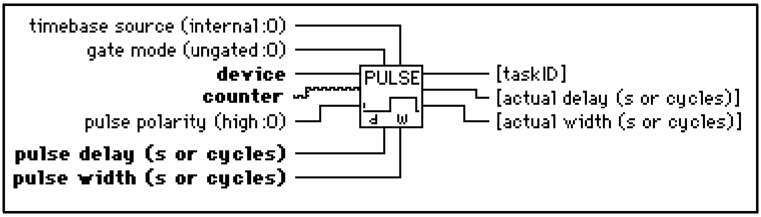
Configures one or two counters to count external events or elapsed time. An external event is a high or low signal transition on the specified SOURCE pin of the counter.



To count events, set the event source/timebase to 0.0 and connect the signal you want to count to the SOURCE pin of the counter. To count time, set this control to the timebase frequency you want to use.

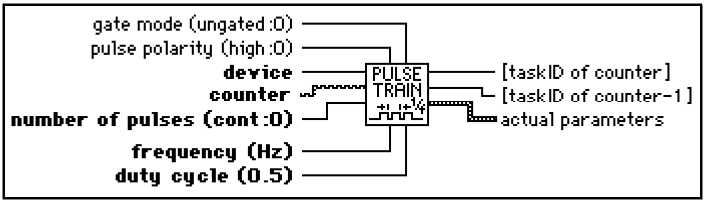
Generate Delayed Pulse

Configures and starts a counter to generate a single pulse with the specified delay and pulse width on the counter's OUT pin. A single pulse consists of a delay phase (phase 1), followed by a pulse phase (phase 2), and than a return to the phase 1 level. If an internal timebase is chosen, the VI selects the highest resolution timebase for the counter to achieve the desired characteristics. If an external timebase signal is chosen, the user indicates the delay and width as cycles of that signal. Execute the Counter Start VI with this VI's taskID to generate another pulse. You can optionally gate or trigger the pulse with a signal on the counter's GATE pin.


timebase source (internal:0)
gate mode (ungated:0)
device
counter
pulse polarity (high:0)
pulse delay (s or cycles)
pulse width (s or cycles)
PULSE
d w
[taskID]
[actual delay (s or cycles)]
[actual width (s or cycles)]

Generate Pulse Train

Configures the specified counter to generate a continuous pulse train on the counter's OUT pin, or to generate a finite-length pulse train using the specified counter and an adjacent counter. The signal has the prescribed frequency, duty cycle, and polarity. Each cycle of the pulse train consist of a delay phase (phase 1) followed by a pulse phase (phase 2).

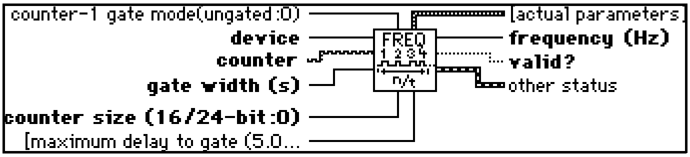


This VI uses only the specified **counter** to generates a continuous pulse. For a finite-length pulse, the VI also uses counter–1 to generate a minimum-delayed pulse to gate **counter**. To generate another pulse train, execute the intermediate Counter Start VI with the **taskID**s supplied by this VI. To stop a continuous pulse train, execute the intermediate Counter Stop VI or execute this counter again to generate one, short pulse. You must externally wire counter–1's OUT pin to **counter's** GATE pin for a finite-length pulse train. You can optionally gate or trigger the start of the train with a signal on the counter–1's GATE pin.

☞ **Note:** ***A pulse train consists of a series of delayed pulses, where phase 1 or the first phase of each pulse is the inactive state of the output (low for a high pulse) and the phase 2 of the second phase is the pulse itself. Refer to the following illustration of a high polarity pulse train.***

Measure Frequency

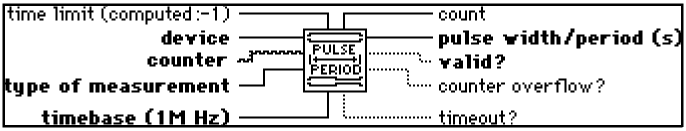
Measures the frequency of a TTL signal on the specified counter's SOURCE pin by counting positive edges of the signal during a specified period of time. In addition to this connection, you must wire the counter's GATE pin to the OUT pin of counter–1. This VI is useful for relatively high frequency signals, when many cycles of the signal occur during the timing period. Use the Measure Pulse Width or Period VI for relatively low frequency signals. Keep in mind that period(s) = 1/frequency (Hz).



This VI configures the specified **counter** and counter+1 (optional) as event counters to count rising edges of the signal on counter's SOURCE pin. The VI also configures counter–1 to generate a minimum-delayed pulse to gate the event counter, starts the event counter and then the gate counter, waits the expected gate period, and then reads the gate counter until its output state is low. Next the VI reads the event counter and computes the signal frequency (**number of events/actual gate pulse width**) and stops the counters. You can optionally gate or trigger the operation with a signal on counter–1’s GATE pin.

Measure Pulse Width or Period

Measures the pulse width (length of time a signal is high or low) or period (length of time between adjacent rising or falling edges) of a TTL signal connected to the counter’s GATE pin. The method used gates an internal timebase clock with the signal being measured. This VI is useful in measuring the period or frequency (1/period) of relatively low frequency signals, when many timebase cycles occur during the gate. Use the Measure Frequency VI to measure the period or frequency of relatively high frequency signals.



The VI iterates until a valid measurement, timeout, or counter overflow occurs. A valid measurement exists when **count** (4 without a counter overflow. If counter overflow occurs, lower the timebase. If you start a pulse width measurement during the phase you want to measure, you get an incorrect low measurement. Therefore, make sure the pulse does not occur until after the counter is started. This restriction does not apply to period measurements.

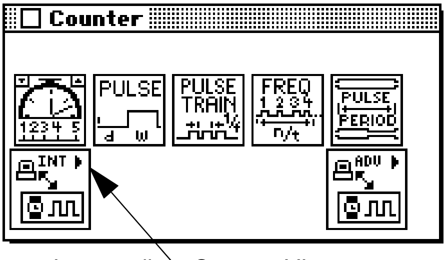
Chapter 26

Intermediate Counter VIs

This chapter describes Intermediate Counter VIs you can use to program counters on MIO, TIO, and other devices with the Am9513 or DAQ-STC counter chips. These VIs call the Advanced Counter VIs to configure the counters for common operations and to start, read, and stop the counters. You can configure these VIs to generate single pulses and continuous pulse trains, to count events or elapsed time, to divide down a signal, and to measure pulse width or period. The Easy Counter VIs call these Intermediate VIs for several pulse generation, counting, and measurement operations.

This chapter also describes the ICTR Control VI that you use with Lab and 1200 Series and PC-LPM devices that contain the 8253 counter/timer chip.

You can access the Intermediate Counter VIs by choosing **Functions»Data Acquisition»Counter**. The Intermediate Counter VIs are the VIs on the second row of the **Counter** palette, as shown below.



Intermediate Counter VIs

Handling Errors

LabVIEW makes error handling easy with the Intermediate Counter VIs. Each intermediate-level VI has an **error in** input cluster and an **error out** output cluster. The clusters contain a Boolean that indicates whether an error occurred, the error code for the error, and the name of the VI that returned the error. If **error in** indicates an error, the VI returns the error information in **error out** and does not continue to run.

When you use any of the Intermediate Counter VIs in a While Loop, you should stop the loop if the **status** in the **error out** cluster reads `TRUE`. If you wire the error cluster to the General Error Handler VI, the VI deciphers the error information and describes the error to you.

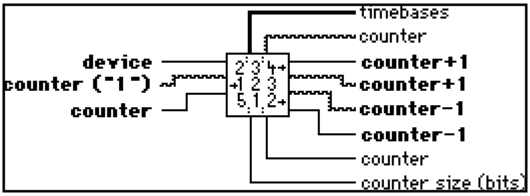
The General Error Handler VI is in **Functions»Utilities** in LabVIEW. For more information on this VI, refer to your *LabVIEW User Manual*.

Intermediate Counter VI Descriptions

The following Intermediate Counter VIs are available.

Adjacent Counters

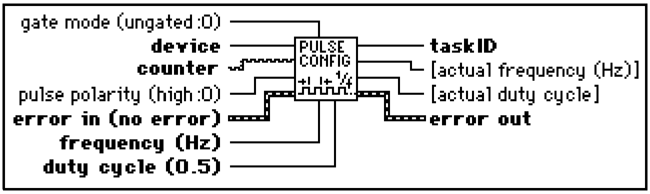
This VI identifies the counters logically adjacent to a specified counter of an MIO or TIO device. It also returns the counter size (number of bits) and the timebases.



Devices with the Am9513 chip have one or two sets of five, 16-bit counters (1–5, 6–10) that can be connected in a circular fashion. For example, the next higher counter to counter 1 (called counter+1) is 2 and the next lower one (called counter–1) is 5.

Continuous Pulse Generator Config

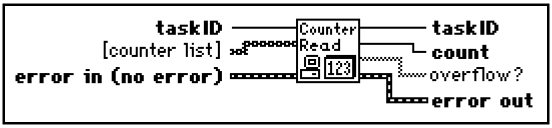
Configures a counter to generate a continuous TTL pulse train on its OUT pin.



The signal is created by repeatedly decrementing the counter twice, first for the delay to the pulse (phase 1), then for the pulse itself (phase two). The VI selects the highest resolution timebase to achieve the desired characteristics. You can optionally gate or trigger the operation with a signal on the counter's GATE pin. Call the Counter Start VI to start the pulse train or to enable it to be gated.

Counter Read

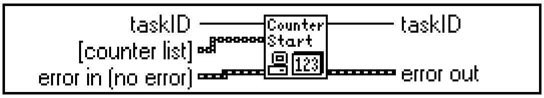
Reads the counter or counters identified by **taskID**.



The VI is designed to read one counter or two concatenated counters of an Am9513 counter chip or to read one counter of a DAQ-STC counter chip.

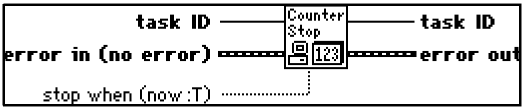
Counter Start

Starts the counters identified by **taskID**.



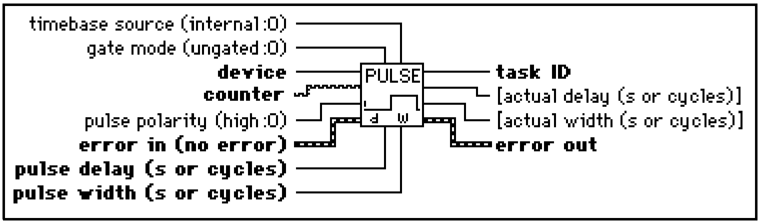
Counter Stop

Stops a count operation immediately or conditionally on an input error.



Delayed Pulse Generator Config

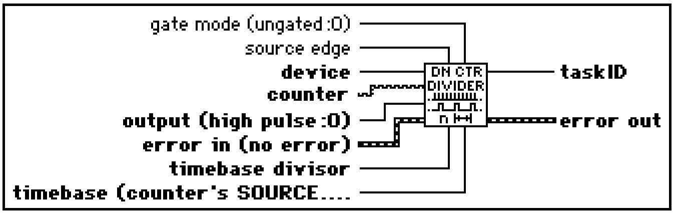
Configures a counter to generate a single, delayed TTL pulse on its OUT pin.


timebase source (internal:0)
gate mode (ungated:0)
device
counter
pulse polarity (high:0)
error in (no error)
pulse delay (s or cycles)
pulse width (s or cycles)
PULSE
d
w
task ID
[actual delay (s or cycles)]
[actual width (s or cycles)]
error out

The signal is created by decrementing the counter twice, first for the delay to the pulse (called phase 1), then for the pulse itself (phase 2). If an internal timebase is chosen, the VI selects the highest resolution timebase for the counter to achieve the desired characteristics. If an external timebase signal is chosen, the user designates the delay and width as cycles of that signal. You can optionally gate or trigger the operation with a signal on the counter's GATE pin. Call the Counter Start VI to start the pulse or enable it to be gated.

Down Counter or Divider Config

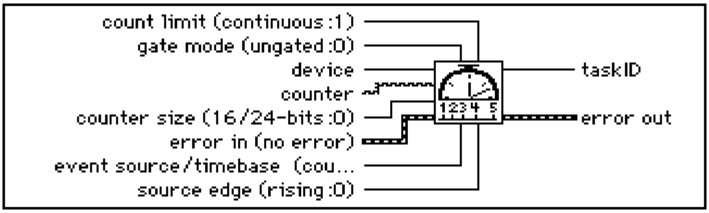
Configures the specified counter to count down or divide a signal on the counter's SOURCE pin or on an internal timebase signal using a count value called the **timebase divisor**. The result is that the signal on the counter's OUT pin is equal to the frequency of the input signal/**timebase divisor**.



You can use this VI to generate finite pulse trains by enabling a continuous pulse generator until the desired number of pulses has occurred. You can also use it in place of the Continuous Pulse Generator Config VI to generate a train of strobe or trigger signals.

Event or Time Counter Config

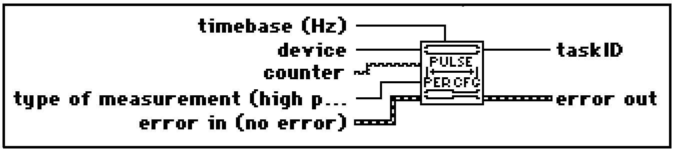
Configures one or two counters to count edges in the signal on the specified counter's SOURCE pin or the number of cycles of a specified internal timebase signal.



When the internal timebase is used, this VI works like the Tick Count (ms) function but uses a hardware counter on the DAQ device with programmable resolution. You can optionally gate or trigger the operation with a signal on the counter's GATE pin. Call the Counter Start VI to start the operation or enable it to be gated.

Pulse Width or Period Meas Config

Configures the specified counter to measure the pulse width or period of a TTL signal connected to its GATE pin.

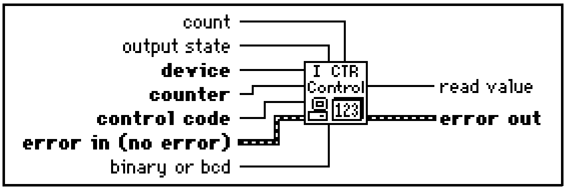


The measurement is done by counting the number of cycles of the specified timebase between the appropriate starting and ending events. To accurately measure pulse width, the pulse must occur after the counter is started. Call the Counter Start VI to start the operation. You can also use this VI to measure the frequency of low frequency signals. For more accurate measurements, use a faster timebase.

ICTR Control

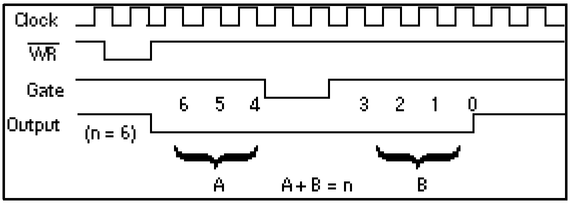
Controls counters the following devices that use the 8253 chip:

- Lab and 1200 Series devices, DAQCard-500, and DAQCard 700
- **(Windows)** LPM devices, 516 devices



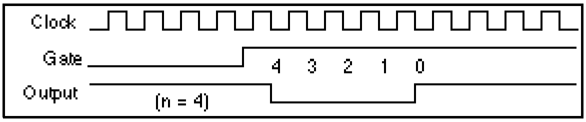
In setup mode 0, as shown in Figure 26-1, the output becomes low after the mode set operation, and the counter begins to count down while the gate input is high. The output becomes high when counter reaches the TC (that is, when the counter decreases to 0) and stays high until you set the selected counter to a different mode.

Figure 26-1.



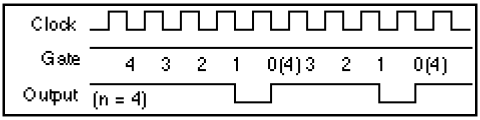
In setup mode 1, as shown in Figure 26-2, the output becomes low on the count following the leading edge of the gate input and becomes high on TC.

Figure 26-2.



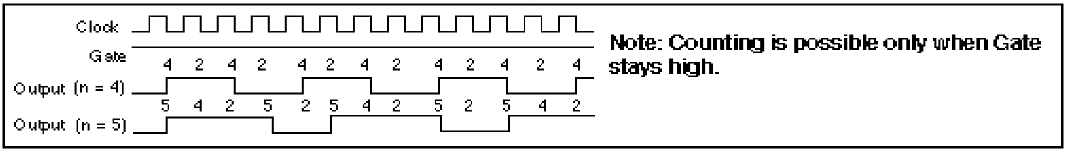
In setup mode 2, as shown in Figure 26-3, the output becomes low for one period of the clock input. The **count** indicates the period between output pulses.

Figure 26-3.



In setup mode 3, the output stays high for one-half of the **count** clock pulses and stays low for the other half. Refer to Figure 26-4.

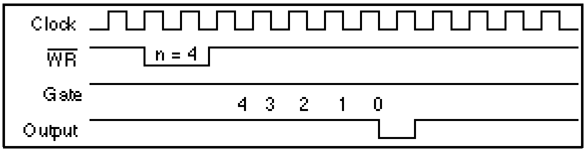
Figure 26-4.



Note: Counting is possible only when Gate stays high.

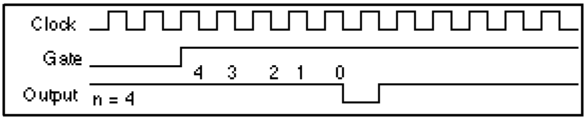
In setup mode 4, as in Figure 26-5, the output is initially high, and the counter begins to count down while the gate input is high. On TC, the output becomes low for one clock pulse, then becomes high again.

Figure 26-5.



Setup mode 5 is similar to mode 4, except that the gate input triggers the count to start. See Figure 26-6 for an illustration of mode 5.

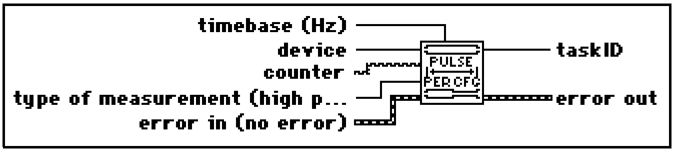
Figure 26-6.



See the 8253 Programmable Interval Timer data sheet in your Lab device user manual for details on these modes and their associated timing diagrams.

Pulse Width or Period Meas Config

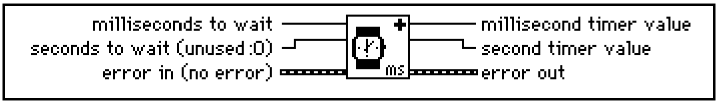
Configures the specified counter to measure the pulse width or period of a TTL signal connected to its GATE pin.



The measurement is done by counting the number of cycles of the specified timebase between the appropriate starting and ending events. To accurately measure pulse width, the pulse must occur after the counter is started. Call the Counter Start VI to start the operation. You can also use this VI to measure the frequency of low frequency signals. For more accurate measurements, use a faster timebase.

Wait+ (ms)

Calls the Wait (ms) function only if no input error exists.



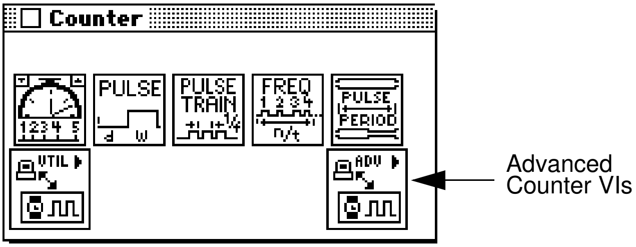
This VI is useful when you want to wait between calls to I/O subVIs that use the error I/O mechanism; without it you need to use a Sequence Structure to control the execution order.

Chapter 27

Advanced Counter VIs

This chapter describes the VIs that configure and control hardware counters. You can use these VIs to generate variable duty cycle square waves, to count events, and to measure periods and frequencies.

You can access the **Advanced Counter** palette by choosing **Functions»Data Acquisition»Counter»Advanced Analog Input**. The icon that you must select to access the Advanced Counter VIs is on the bottom row of the **Counter** palette, as shown below.



☞ **Note:** ***An important basic data acquisition concept is to use only the inputs that you need on each VI. Leave the rest of the inputs unwired, and LabVIEW sets them to their default values. In the Help window, the most important terminals are labeled in bold, and the least commonly used are in brackets. Values given in parentheses are default values.***

The following lists the type of counter chips that your device must have to work with your version of LabVIEW:

- Am9513, 8253, or DAQ-STC Counter Chip
- DAQ-STC Counter Chip

The ICTRControl VI works with devices that contain the 8253 counter chip.

Refer to Table 27-1 for the counter chips used with the various devices.

Table 27-1. Counter Chips and Their Available DAQ Devices

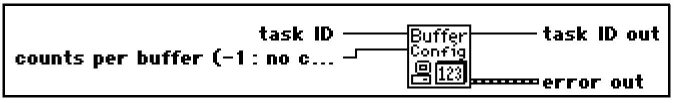
Counter Chip	DAQ Device
Am9513	AT-MIO-16, AT-MIO-16D, AT-MIO-16F-5, AT-MIO-16X, AT-MIO-64F-5, PC-TIO-10, All AO-2DC Devices, EISA-A2000, NB-MIO-16, NB-MIO-16X, NB-DMA-8-G, NB-DMA2800, NB-TIO-10, NB-A2000
DAQ-STC	All E Series Devices, 5102 Devices
8253	All Lab and 1200 Series Devices, DAQCard-500, DAQCard-700, LPM Devices, 516 Devices

Advanced Counter VI Descriptions

The following Advanced Counter VIs are available.

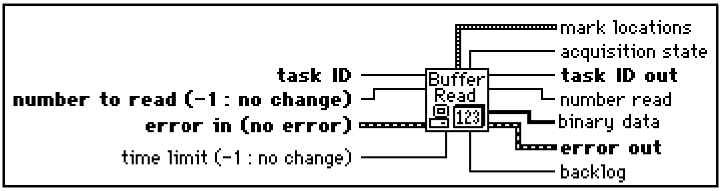
CTR Buffer Config

Allocates memory where LabVIEW stores counter data. The CTR Buffer Config VI also configures the specified group to perform buffered counter operations instead of the normal single point operations.



CTR Buffer Read

Returns data from the buffer allocated by CTR Buffer Config.



Note: *Incremental reading from the count buffer is not supported at this time. Therefore, you must allow the buffer to fill before you read from it and then you must read all of it. Until incremental reading and circular use of the buffer are implemented, leave* **number to read** *unwired (with a value of −1) or set it to the value of* **counts per buffer.**

CTR Group Config

Collects one or more counters into a group. You can use counter groups containing more than one counter to start, stop, or read multiple counters simultaneously. DAQ-STC devices do not currently support multiple counter groups.

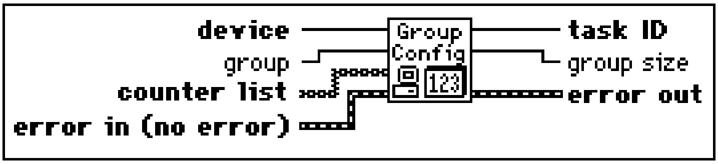


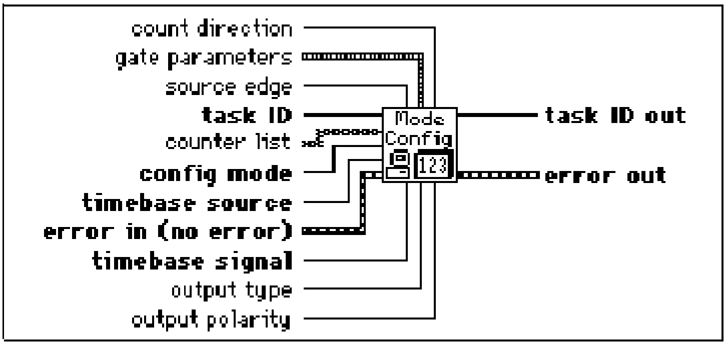
Table 27-2 contains valid counter numbers for devices supported by this VI.

Table 27-2. Valid Counter Numbers for CTR Group Config Devices

Device Type	Valid Numbers
DAQ-STC Devices	0 and 1
Am9513 MIO Devices	1, 2, and 5
NB-DMA-8-G, NB-DMA2800	1 through 5
PC-TIO-10, NB-TIO-10	1 through 10
EISA-A2000, NB-A2000	2

CTR Mode Config

Configures one or more counters for a designated counter operation and selects the source signal, gating mode, and output behavior on terminal count (TC).



This VI does not start the counters. Use CTR Control VI with **control code** 1 (Start) to start the counters. If you are using a counter for pulse generation, you do not have to call this VI unless you want to change the gate mode or output behavior.

Modes 3, 4, and 6 can be used with or without buffered counting. Mode 7 must be used with buffered counting. With buffered counting, call the CTR Buffer Config VI before or after the CTR Mode Config VI and before the CTR Control VI to start the operation, then call the CTR Buffer Read VI to read the buffered count values. With buffered or unbuffered operations, call the CTR Control VI to read the most recently acquired, unbuffered count value.

Unless otherwise stated, the following figures show timing and counter values for operations in which the **gate mode** is set to high-level or rising-edge and the **source edge** is set to rising-edge.

Use mode 1 to reset all the CTR Mode Config VI parameters to their default settings. This mode overrides any conflicting parameter settings.

Use mode 2 to count transitions of the selected signal and to stop at the first TC. The overflow status bit is set at TC. Use the CTR Control VI to read the overflow status. This mode is available only with Am9513 devices. Mode 2 counting is unbuffered.

Figure 27-1 shows the count values you would read with this mode using three **gate mode** settings (gating off; high-level gating; and rising-edge gating).

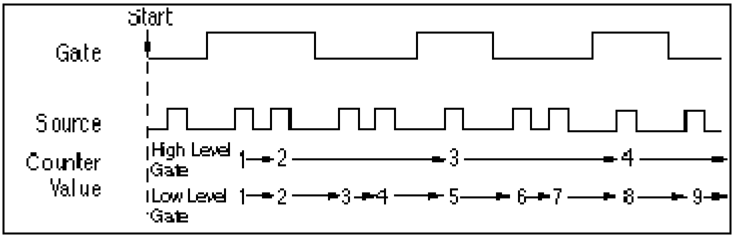


Figure 27-1. Unbuffered Mode 2 and 3 Counting

Use mode 3 to count transitions of the selected signal continuously, rolling over at TC and then continuing on. Figure 27-1 shows unbuffered mode 3 counting. Figure 27-2 illustrates a buffered mode 3 operation with rising-edge gating. This buffered operation is available only with DAQ-STC devices. With buffered mode 3 operation, LabVIEW stores the current count value into the buffer on each selected edge of the source signal.

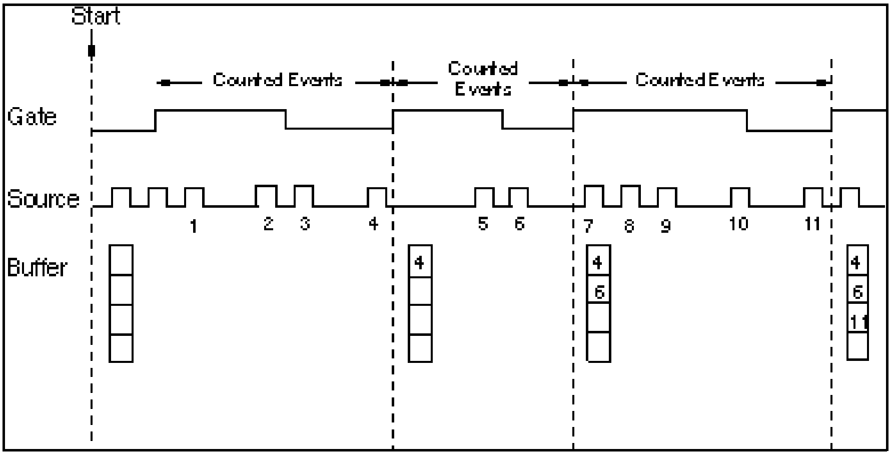


Figure 27-2. Buffered Mode 3 Counting

Use mode 4 with level gating to measure pulse width and with edge gating to measure the period of the selected gate signal.

Note: ***For the following descriptions of pulse width measurements (modes 4, 6, and 7), a high pulse is defined simply as the high-level phase of a signal when*** **gate mode** ***is set to high-level gating. This definition differs from that of a high pulse using pulse generation (mode 5), which consists of a low***

level delay phase followed by a high level pulse phase. (Low pulses are similarly defined by switching the words high and low.)

To measure pulse width, set the **gate mode** to high or low level. Figure 27-3 shows unbuffered mode 4 pulse width measurements. You can start an Am9513 counter at any time, and it will measure pulses until you stop it. If you start it in the middle of the pulse you want to measure (for example, during a high pulse for high-level gating), LabVIEW returns a short count for that measurement. You must start a DAQ-STC counter only when the signal is in the opposite polarity from the selected gate level (for example, a low-level phase for high-level gating). Otherwise, the VI returns error number –10890. With unbuffered counting, the DAQ-STC stops counting after one measurement. Mode 5 configures the counter for pulse generation. Use the CTR Pulse Config VI to specify the pulse you want to generate.

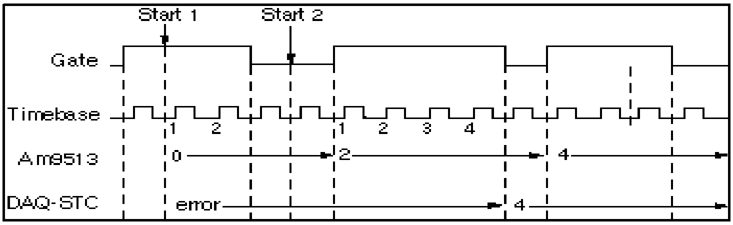


Figure 27-3. Unbuffered Mode 4 High Pulse Width Measurement

Figure 27-4 shows the buffered mode 4 pulse width measurement, which is available only with DAQ-STC devices. The measured value is stored into the buffer at the end of each pulse. See mode 6 for another way to measure pulse width with a DAQ-STC device.

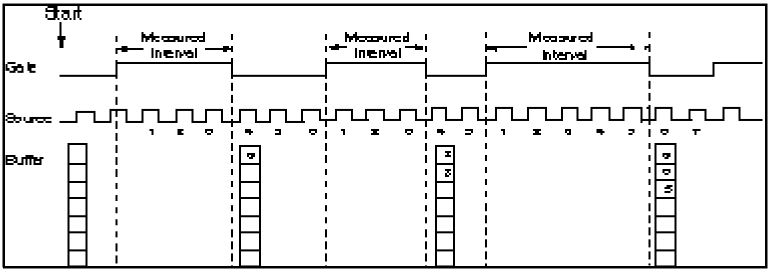


Figure 27-4. Buffered Mode 4 Rising-Edge Pulse Width Measurement

To measure period, set the **gate mode** to rising or falling edge. Figure 27-5 shows unbuffered mode 4 pulse width measurement.

You may start either an Am9513 or a DAQ-STC counter at any time. The counter begins counting at the start of the next period. The Am9513 counter measures periods continuously. With unbuffered counting, the DAQ-STC stops counting after one measurement.

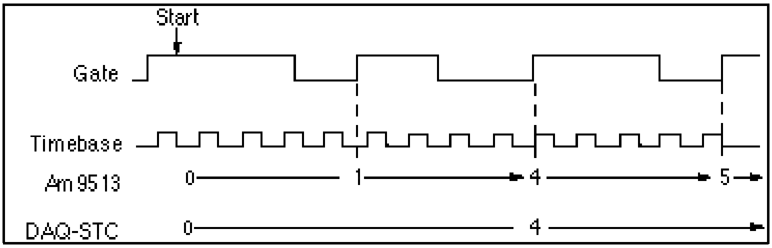


Figure 27-5. Unbuffered Mode 4 Rising-Edge Period Measurement

Figure 27-6 shows buffered mode 4 period measurement, which is available only with DAQ-STC devices. The measured value is stored into the buffer at the end of each period.

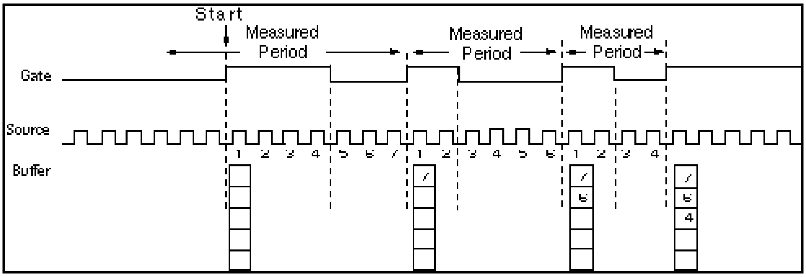


Figure 27-6. Buffered Mode 4 Rising-Edge Pulse Width Measurement

Use mode 5 to configure for pulse generation when you also need to configure the **gate mode**, **output type**, or **output polarity** to non-default values. Otherwise, avoid calling the CTR Mode Config VI and use only the CTR Pulse Config VI for pulse generation. See the CTR Pulse Config VI more additional information about this operation.

Use mode 6 with level gating to measure the pulse width of the selected signal. This mode is available only with DAQ-STC devices. Mode 6 differs from mode 4 in that the measurement of a high (low) pulse does not begin until the first falling (rising) edge of the signal after you start the counter. If you use unbuffered counting, the counter continues to measure pulses until you call the CTR Control VI to read the most recently measured value, at which time the counter stops. Unbuffered mode 6 counting is illustrated in Figure 27-7.

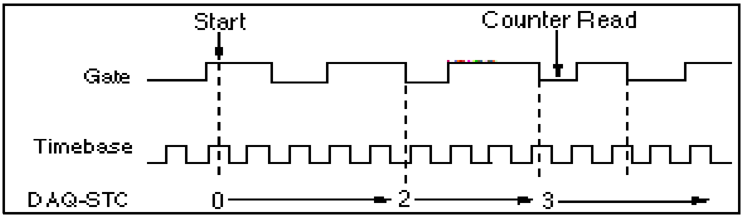


Figure 27-7. Unbuffered Mode 6 High Pulse Width Measurement

With buffered mode 6 counting, the measured value is stored into the buffer at the end of each pulse, as illustrated with Figure 27-8. Call the CTR Buffer Read VI to read the values.

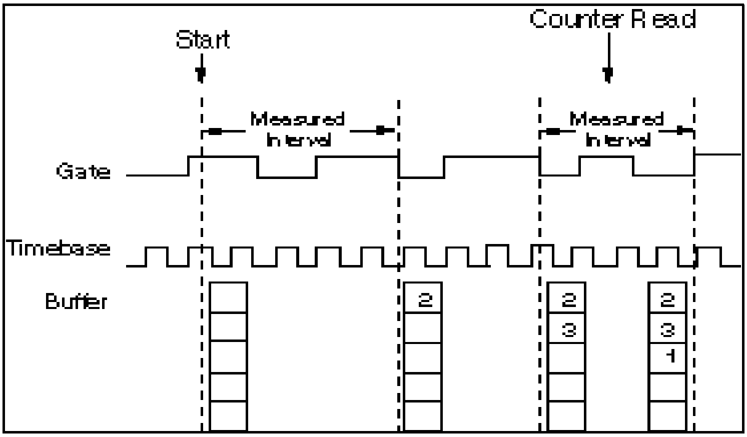


Figure 27-8. Buffered Mode 6 High Pulse Width Measurement (Count on Rising Edge of Source)

Use mode 7 to measure every phase of the selected signal using buffered counting. This mode is available only with DAQ-STC devices. The count value is stored in the buffer on each low-to-high and high-to-low transition. Use the CTR Buffer Read VI to read the values. To measure period with this mode, sum successive pairs of signals. To measure phase, use every other value. LabVIEW ignores the value of **gate mode** with mode 7, which means that you cannot tell whether the first measurement starts at a rising or falling edge.

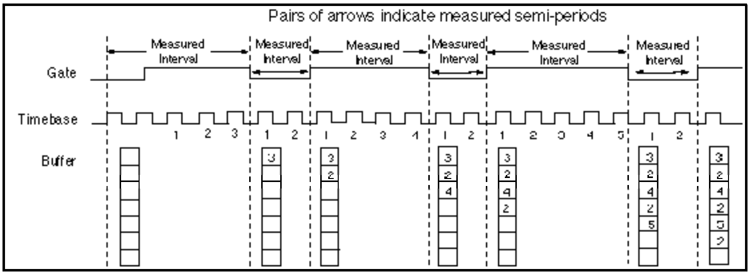


Figure 27-9. Buffered Mode 7 Semi-Period Measurement

Table 27-3 shows the legal values and default settings for **timebase signal**. A value of `-1` tells LabVIEW to use the default settings. When the table says counter, it refers to the counter being configured. If there are multiple counters, LabVIEW configures each counter successively.

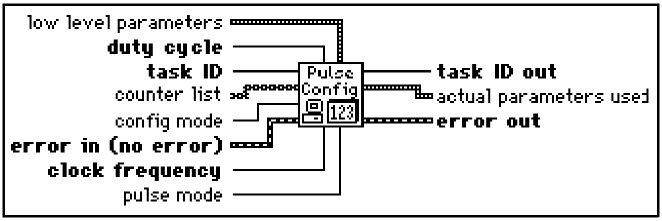
Refer to Table 27-3 to determine what is the next higher or lower consecutive counter.

Table 27-3. Adjacent Counters.

Device Type	Next Lower Counter	Counter	Next Higher Counter
Am9513	5	1	2
	1	2	3
	2	3	4
	3	4	5
	4	5	1
	10	6	7
	6	7	8
	7	8	9
	8	9	10
	9	10	6
DAQ-STC	1	0	1
	0	1	0

CTR Pulse Config

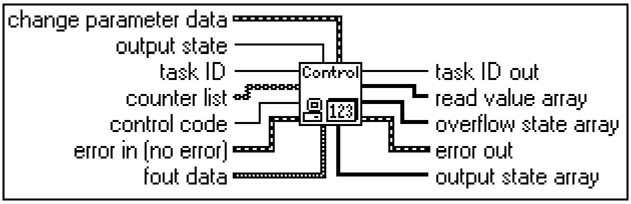
Specifies the parameters for pulse generation. This VI configures the counters but does not start them. Use the CTR Control VI with control code `1` (Start) to produce the pulse.



Use this VI to specify the characteristics of your pulses. You can also use the CTR Mode Config VI to set your desired gate modes, output polarity, and output type. Use the CTR Pulse Config VI to specify **timebase source** and **timebase signal** for pulse generation, because LabVIEW ignores these values specified in the CTR Mode Config VI.

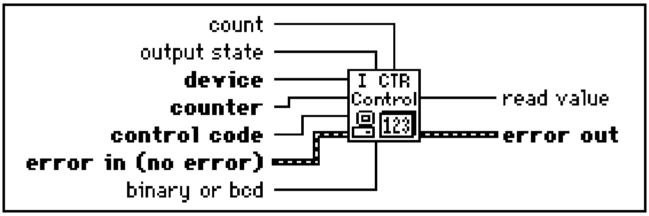
CTR Control

Controls and reads groups of counters. Control operations include starting, stopping, and setting the output state.



ICTRControl

Controls counters on devices that use the 8253 chip (Lab and 1200 Series devices, 516_devices PC-LPM-16, DAQCard-500, and DAQCard 700).



Chapter 28

Calibration and Configuration VIs

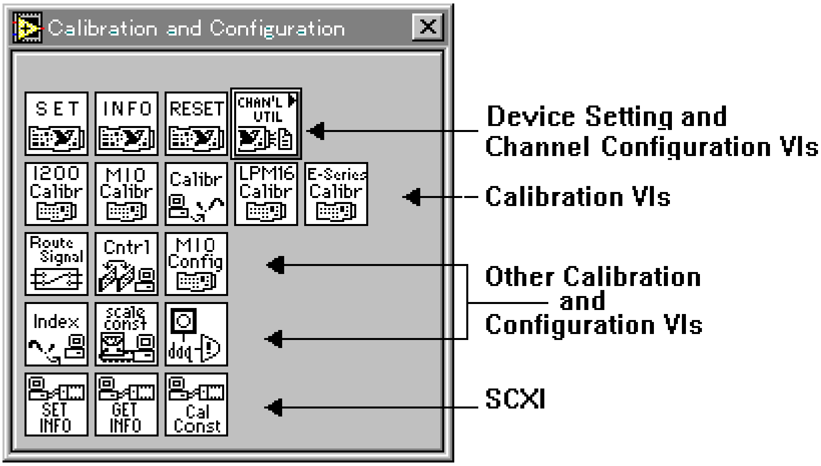
This chapter describes the VIs that calibrate specific devices and set and return configuration information.

This chapter also includes a VI for controlling the RTSI bus, which is a triggering and timing bus you can use to synchronize, time, and trigger multiple DAQ devices.

(Windows) There is also a VI you can use to set up data acquisition event occurrences.

You can calibrate certain DAQ devices with the device-specific VIs, but this is not always necessary because National Instruments calibrates all devices at the factory.

You can access the Calibration and Configuration VIs by choosing **Functions»Data Acquisition»Calibration and Configuration** as shown below.

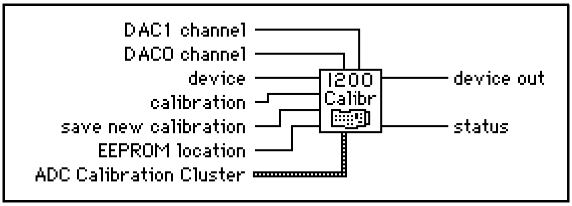


Calibration and Configuration VI Descriptions

The following Calibration and Configuration VIs are available.

1200 Calibrate

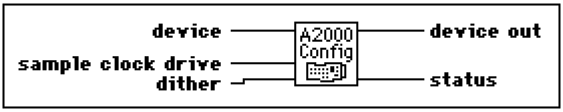
This VI calibrates the gain and offset values for the ADCs and DACs on 1200 Series devices (i.e., DAQPad-1200, DAQCard-1200, etc.).



You can perform a new calibration (and optionally save the new calibration constants in one of four user areas in the onboard EEPROM) or load an existing set of calibration constants by copying them from their storage location in the onboard EEPROM. LabVIEW automatically loads the calibration constants stored in the onboard EEPROM load area when LabVIEW launches or when you reset the device. By default the EEPROM load area contains a copy of the calibration constants in the factory area

A2000 Calibrate

Calibrates the NB-A2000 or EISA-A2000 A/D gain and offset values or restores them to the original factory-set values.



You can calibrate your NB-A2000 or EISA-A2000 to adjust the accuracy of the readings from the four analog input channels. LabVIEW automatically loads the stored calibration values when it launches or when you reset your NB-A2000 or EISA-A2000.

Warning: ***This VI is supported only up to NI-DAQ version 4.9.0 and has been removed from the Calibration and Configuration palette. This VI is still included in the DAQ VI Library for compatibility only, therefore if you are using NI-DAQ version 5.0 or later, this VI will return the following message:*** `deviceSupportError`**.** ***If you wish to use this VI, please re-install NI-DAQ version 4.9.0 or an earlier version.***

Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the NB-A2000 or EISA-A2000 DAQ devices.

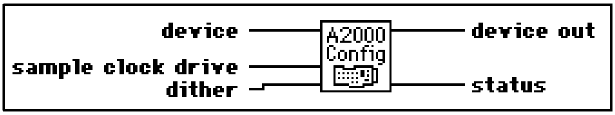
Warning: ***Read the calibration chapter in the*** *NB-A2000 or EISA-A2000 User Manual* ***before using the A2000 Calibrate VI.***

If you set **save new values** to `1`, then this VI stores the gain and offset calibration values in an EEPROM on the NB-A2000 or EISA-A2000 device, which does not lose its data even if the device loses power. LabVIEW reads these EEPROM values and loads them into the NB-A2000 or EISA-A2000, you can choose to replace the permanent copies of the gain and offset EEPROM values and use the new values until the next calibration, even if you reinitialize the device. You can also choose not to replace the EEPROM values, but to use the new values until the next calibration or initialization.

For example, if you consistently get inaccurate readings from one or more input channels after you reset the device, you can calibrate and save the new gain and offset values as permanent copies in the EEPROM. However, if acquisition results are accurate after initialization but start to drift after a few hours of device operation when the device temperature increases, you can calibrate the device at this operating temperature and retain the current EEPROM values to use after the next initialization.

A2000 Configure

Configures dithering and whether to drive the SAMPCLK* line for the NB-A2000 or EISA-A2000.



Warning: ***This VI is supported only up to NI-DAQ version 4.9.0 and has been removed from the Calibration and Configuration palette. This VI is still included in the DAQ VI Library for compatibility only, therefore if you are using NI-DAQ version 5.0 or later, this VI will return the following message:*** `deviceSupportError`***. If you wish to use this VI, please re-install NI-DAQ version 4.9.0 or an earlier version.***

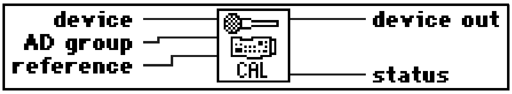
Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the NB-A2000 or EISA-A2000 DAQ devices.

After system startup, LabVIEW configures the NB-A2000 or EISA-A2000 as follows.

- **sample clock drive** = `0`: Sample clock signal does not drive SAMPCLK* line.
- **dither** = `0`: Dither disabled.

A2100 Calibrate (Macintosh)

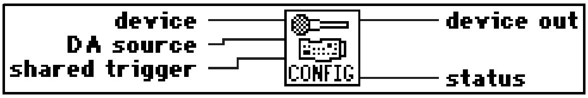
Selects the desired calibration reference and performs an offset calibration cycle on the ADCs on the NB-A2100 or the NB-A2150.



NI-DAQ driver software calibrates the two A/D channels using the analog input ground as the reference for each channel when you turn on the computer.

A2100 Config (Macintosh)

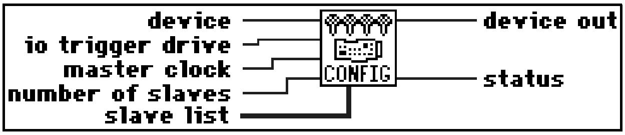
Selects the signal source used to provide data to the DACs and lets you configure the external digital trigger to be shared by data acquisition and waveform generation operations on the NB-A2100.



If LabVIEW acquires multiple data acquisition frames and generates multiple waveform cycles with a trigger required at the beginning of each cycle, then the external trigger recognition synchronizes so that each trigger simultaneously initiates the acquisition of the next data frame while generating the output of the next waveform cycle.

A2150 Config (Macintosh)

Selects whether or not LabVIEW should drive an internally generated trigger to the NB-A2150 I/O connector. This VI also determines whether LabVIEW should drive the NB-A2150 sampling clock signal over the RTSI bus to other devices for multiple-device synchronized data acquisition.



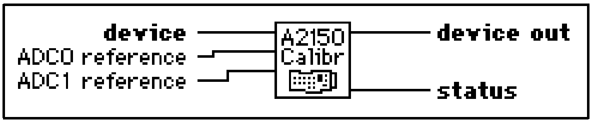
Enable **io trigger drive** only if you have executed the RTSI Control VI to receive the RTSITRIG* signal over the RTSI bus, or if you have enabled the analog level trigger using the AI Trigger Config VI. In these cases, you can monitor the signal being sent to the A/D trigger circuitry at the EXTTRIG* line of the I/O connector after starting the acquisition. A high-to-low edge of the signal triggers the data acquisition.

The NB-A2150 uses signals over the RTSI bus for sampling clock synchronization between two or more NB-A2150 devices. The sampling clock synchronization circuitry makes simultaneous sampling possible on more than four channels using additional NB-A2150 devices. If **master clock** is `1`, **slave list** should contain the list of devices that accept the sampling clock from **device**. After you run A2150 Config with **master clock** equal to `1` and **number of slaves** greater than `0`, you cannot use the AI Clock Config to set the scan rate for devices in **slave list** until you run A2150 Config again on **device** with **master clock** equal to `1` and **number of slaves** equal to `0`.

☞ **Note:** ***Executing A2150 Config with master clock equal to 1 and number of slaves equal to 0 deconfigures the devices previously in the slave list and sets them up to use their own sampling clock signal.***

A2150 Calibrate (Windows)

Performs offset calibrations on the ADCs of the specified AT-A2150.

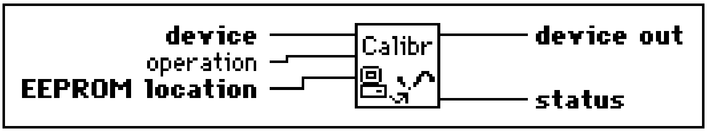


Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the AT-A2150 DAQ device.

When you launch LabVIEW, or when you reset the AT-A2150, LabVIEW performs an offset calibration using the analog ground as the reference. Use this VI only for device calibration to an external reference or for device recalibration for ground reference after using an external reference.

AO-6/10 Calibrate (Windows)

Loads a set of calibration constants into the calibration DACs or copies a set of calibration constants from one of four EEPROM areas to EEPROM area 1.



You can load an existing set of calibration constants into the calibration DACs from a storage area in the onboard EEPROM. You can copy EEPROM storage areas 2 through 5 to storage area 1. EEPROM area 5 contains the factory calibration constants. LabVIEW automatically loads the calibration constants stored in EEPROM area 1 upon start-up or when you reset the AT-AO-6/10.

☞ **Note:** ***You can also use the calibration utility provided with the AT-AO-6/10 to perform a calibration procedure. Refer to the calibration chapter in the*** *AT-AO-6/10 User Manual* ***for more information.***

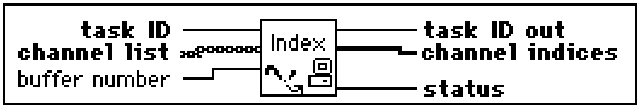
Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the AT-AO-6/10 DAQ devices.

When LabVIEW initializes the AT-AO-6/10, the DAC calibration constants stored in **EEPROM location** 1 (user calibration area 1) provide the gain and offset values that ensure proper device operation. So, this initialization is the same as running the AO-6/10 Calibrate VI with **operation** set to `1` and **EEPROM location** set to `1`. When the AT-AO-6/10 leaves the factory, **EEPROM location** 1 contains a copy of the calibration constants stored in **EEPROM location** 5 (factory calibration).

A calibration procedure performed in bipolar mode is not valid for unipolar mode and vice versa. See the calibration chapter of the *AT-AO-6/10 User Manual* for more information.

Channel To Index

Uses the current group configuration for the specified task to produce a list of indices into the group's scan or update list for each channel specified in the channel list.



You can use this list of channel indices to locate data for a particular channel within a multiple channel buffer. You can also use the indices to read or write to a group subset with the buffer read and write VIs.

Refer to your specific device information in Appendix A, *DAQ Hardware Capabilities*, for the channel limitations that apply to your device.

Table 28-1 shows possible values for the **channel scan list**, **channel list**, and **channel indices** parameters. Table 28-2 shows the possible values for the Sun. The **channel scan list** parameter is an input for the group configuration VIs.

Table 28-1. Channel to Index VI Parameter Examples

Channel Scan List	Channel List	Channel Indices
`1, 3, 4, 5, 7`	**channel lis**t[0] = `5`	**channel indices**[0] = `3`. Data for channel 5 is at position 3 within a scan. Indices are zero-based.
`1, 3, 4, 5, 7`	**channel list** is of 0 length.	**channel indices** is of 0 length. (In this case, status is non-zero.)
`1, 2, 1, 3, 1, 4` (The device samples channel 1 three times during a scan.)	**channel list**[0] = `1, 1, 1`	**channelindices**[0] = `0`, **channelindices**[1] = `2`, and **channelindices**[2] = `4`. The first occurrence of channel 1 within a scan is at index 0, the second at index 2, and the third at index 4
`0, 1, 3, 4` (For this example, **channel scan list** is a digital input group.)	**channel list**[0] = `3`	**channel indices**[0] = `2`. The eight bits of data from port 3 are at index 2 in the scan list.
`0:3` (One AMUX-64T in use.)	**channel list**[0] = `AM1!9`	**channel indices**[0] = `9`. Data obtained from channel 9 on AMUX-64T device number 1 is at index 9 in the data buffer.

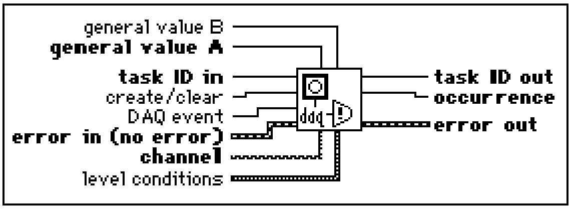
`SC1!MD1!CH0:7,`
`SC1!MD2!CH0:4`
	channel list[0] = `SC1!MD2!CH3`	**channel indices**[0] = `11`.
		Data obtained from channel 3 of the SCXI module in slot 2 is at index 11 in the data buffer.

Table 28-2. Channel to Index VI Parameter Examples for Sun

channel scan list	channel list	channel indices
`1, 3, 4, 5, 7`	**channel list**[0] = `5`	**channel indices**[0] = `3`. Data for channel 5 is at position 3 within a scan. Indices are zero-based.
`1, 3, 4, 5, 7`	**channel list** is of 0 length.	**channel indices** is of 0 length. (In this case, status is non-zero.)
`1, 2, 1, 3, 1, 4` (The device samples channel 1 three times during a scan.)	**channel list**[0] = `1, 1, 1`	**channel indices**[0] = `0`, **channel indices**[1] = `2`, and **channel indices**[2] = `4`. The first occurrence of channel 1 within a scan is at index 0, the second at index 2, and the third at index 4

DAQ Occurrence Config (Windows)

Creates occurrences that are set by data acquisition events.



A DAQ event can be the completion of an acquisition, the acquisition of a certain number of scans, an analog signal meeting certain trigger conditions, a periodic event, an aperiodic (externally driven) event, or a digital pattern match or mismatch. Your VI can sleep while waiting for an occurrence to be set, freeing your computer to execute other VIs.

When you set the **create/clear** control to `1` (create) and call the VI, this VI creates an occurrence. Use the **DAQ event** control to select the event that sets the occurrence. Wire the occurrence this VI produces to the Wait on Occurrence function. Anything you wire to the output of the Wait on Occurrence function does not execute until the occurrence is set. The occurrence is set each time the event occurs. The occurrence does not clear until you set the **create/clear** control to `0` (clear) and call this VI, or call the Device Reset VI for the device.

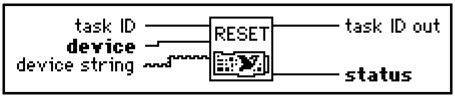
LabVIEW returns a Not a Refnum file I/O constant along with a non-zero status code if it cannot create the occurrence.

For each computer platform, LabVIEW limits the number of occurrences per second that you can set. Although this limit depends on the speed of your computer, avoid exceeding 500 occurrences per second.

For some of the events, you must perform your operation using interrupts instead of DMA. Refer to the description of the **DAQ event** control in this section for more information.

Device Reset

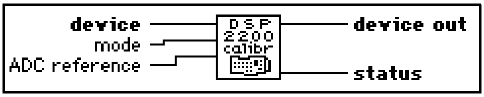
Resets either an entire device or the particular function identified by **taskID**.



Resetting a **taskID** function has the same result as calling the control VI for that function with **control code** set to clear. When you reset the entire device, LabVIEW clears all tasks and changes all device settings to their default values.

DSP2200 Calibrate (Windows)

Performs offset calibrations on the analog input and/or analog output of the AT-DSP2200.



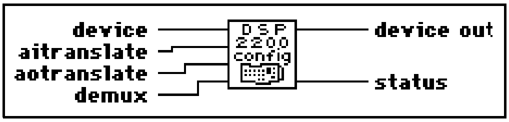
Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the AT-DSP2200 DAQ device.

When you launch LabVIEW or reset the AT-DSP2200, LabVIEW performs an offset calibration on both the analog input and output using analog ground as the reference.

You can use this VI to calibrate the analog input using an external reference or to recalibrate the AT-DSP2200 to compensate for configuration or environmental changes.

DSP2200 Configure (Windows)

Specifies data translation and demultiplexing operations that the AT-DSP2200 performs on analog input and output data.



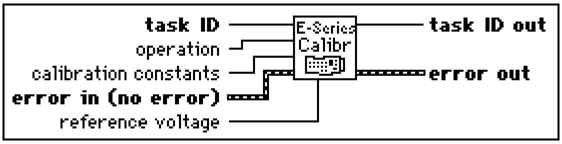
Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the AT-DSP2200 DAQ device.

Because software running locally on the AT&T WE DSP32C DSP chip reads data from the ADCs and writes data to the DACs, you can manipulate the data during these transfers. When you write analog input data to DSP memory, you can write the data as unscaled 16-bit integers, unscaled 32C floating-point numbers, or scaled 32C floating-point voltages. You can use the **demux** option only when you write analog input data to DSP memory. When you enable **demux**, the device writes data from channel 0 consecutively into DSP memory, beginning at the start of each buffer, and writes channel 1 data consecutively beginning at the half-way point of each buffer. When the device writes analog input data to PC memory, it can write the data as unscaled 16-bit integers, unscaled IEEE single-precision floating-point numbers, or scaled IEEE single-precision voltages.

The analog output translations in the opposite directions from the analog input translations. If **aotranslate** is `0`, the source data must be in a format suitable for the DACs (16-bit integer DAC values). If **aotranslate** is `1` or `3`, the source data are DAC values in 32C format in DSP memory or in IEEE single-precision format in PC memory. If **aotranslate** is `2` or `4`, the source data are voltages in 32C format in DSP memory or in IEEE single-precision format in PC memory.

E-Series Calibrate (Windows)

Use this VI to calibrate your E Series device and to select a set of calibration constants to be used by LabVIEW.



Warning: ***Read the calibration chapter in your device user manual before using the E-Series Calibrate VI.***

Your device contains calibration D/A converters (calDACs) that are used for fine-tuning the analog circuitry. The calDACs must be programmed (loaded) with certain numbers, called *calibration constants*. Those constants are stored in non-volatile memory (EEPROM) on your device or are maintained by LabVIEW. To achieve specification accuracy, you should perform an internal calibration of your device just before a measurement session, but after your computer and the device have been powered on and allowed to warm up for at least 15 minutes. Frequent calibration produces the most stable and repeatable measurement performance. The device is not harmed in any way if you recalibrate it as often as you like.

Two sets of calibration constants can reside in two areas inside the EEPROM, called *load areas*. One set of constants is programmed at the factory, the other is left for the user. One load area in the EEPROM corresponds to one set of constants. The load area LabVIEW uses for loading calDACs with calibration constants is called the default load areas. When you get the device from the factory, the default load area is the area that contains the calibration constants obtained by calibrating the device in the factory. LabVIEW automatically loads the relevant calibration constants stored in the load area the first time you call a VI that requires them.

Note: ***Calibration of your E Series device takes some time. Do not be alarmed if the VI takes several seconds to execute.***

Warning: ***When you run this VI with the*** **operation** ***set to self calibrate or external calibrate, LabVIEW will abort any ongoing operations the device is performing and set all configurations to their defaults. Therefore, you should run this VI before any other DAQ VIs or when no other operations are running.***

12-bit E Series Devices

- Connect the positive output of your reference voltage source to the analog input channel 8.
- Connect the negative output of your reference voltage source to the AISENSE line.
- Connect DAC0 line (analog output channel 0) with analog input channel 0.
- If your reference voltage source and your computer are floating with respect to each other, connect the AISENSE line with the AIGND line as well as with the negative output of your reference voltage source.

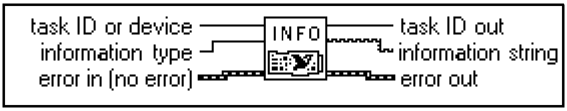
16-bit E Series Devices

- Connect the positive output of your reference voltage source to the analog input channel 0.

- Connect the negative output of your reference voltage source to the analog output channel 8 (by performing those two connections you supply reference voltage to the analog input channel 0, which is configured for differential operation.)
- If your reference voltage source and your computer are floating with respect to each other, connect the negative output of your reference voltage source to the AIGND line, as well as to the analog input channel 8.

Get DAQ Device Information

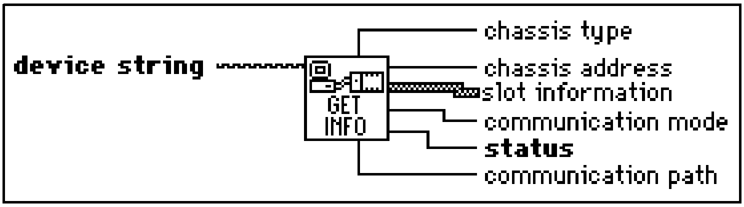
Returns information about a DAQ device.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the transfer methods available with your DAQ device.

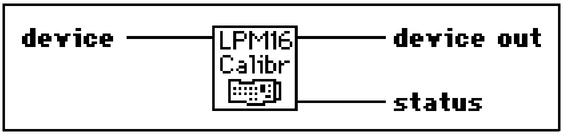
Get SCXI Information

Returns the SCXI chassis configuration information that you set using the configuration utility or the Set SCXI Information VI.



LPM-16 Calibrate

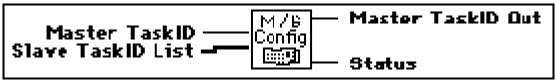
Calibrates the PC-LPM-16 or PC-LPM-16PnP converter. The calibration calculates the correct offset voltage for the voltage comparator, adjusts positive linearity and full-scale errors to less than ±0.5 LSB each, and adjusts zero error to less than ±1 LSB.



Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the PC-LPM-16, DAQCard-500, or DAQCard-700 device.

Master Slave Config

Configures one device as a master device and any remaining devices as slave devices for multiple-buffered analog input operations.



Warning: ***This VI is supported only up to NI-DAQ version 4.9.0 and has been removed from the Calibration and Configuration palette. This VI is still included in the DAQ VI Library for compatibility only, therefore if you are using NI-DAQ version 5.0 or later, this VI will return the following message:*** `deviceSupportError`***. If you wish to use this VI, please re-install NI-DAQ version 4.9.0 or an earlier version.***

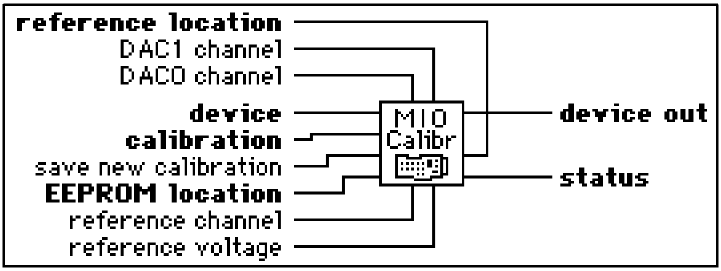
Makes sure LabVIEW always re-enables the slave devices *before* the master device in a multiple-buffer analog input operation. Only the following devices, which support multiple buffered acquisitions, can use this VI.

- **(Macintosh)** NB-A2000, NB-A2100, and NB-A2150.

The master device sends a trigger or clock signal to the slave device(s) to control the slave device sampling. In a multiple-buffer acquisition, you must enable the slave device before the master device to make sure the slave device always responds to a master signal. If you enable the master device first, it can send a signal to the slave devices before they can respond. You are responsible for the initial startup order. You should always start the master device last. The Master Slave Configuration VI makes sure LabVIEW arms the master device last for each subsequent buffer acquired during a multiple-buffer acquisition.

MIO Calibrate (Windows)

Calibrates the AT-MIO-16F-5, AT-MIO-64F-5, and AT-MIO-16X gain and offset values for the ADCs and the DACs. You can either perform a new calibration or use an existing set of calibration constants by copying the constants from their storage location in the onboard EEPROM. You can store several sets of calibration constants. LabVIEW automatically loads the calibration constants stored in the EEPROM load area during startup or when you reset the device.



The load area for the AT-MIO-16F-5 is user area 5. The load area for the AT-MIO-64F-5 and AT-MIO-16X is user area 8.

Warning: ***Read the calibration chapter in your device user manual before using the MIO Calibrate VI.***

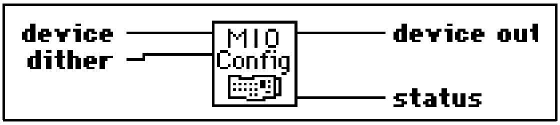
Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the AT-MIO-16F-5, AT-MIO-64F-5, and AT-MIO-16X DAQ devices.

Note: ***You should always calibrate the ADC and the DACs after you calibrate the internal reference voltage.***

Note: ***If the device takes analog input measurements with the wrong set of calibration constants loaded, you may get erroneous data.***

MIO Configure (Windows)

Turns dithering on and off. This VI supports the following devices: AT-MIO-16F-5, AT-MIO-64F-5, all 12-bit E Series devices, and all 1200 Series devices.

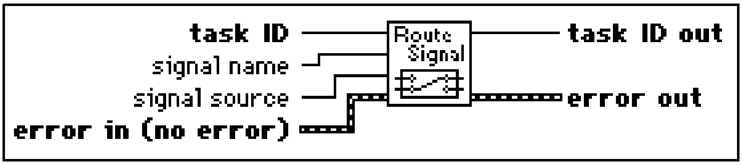


Refer to Appendix A, *DAQ Hardware Capabilities*, for more information on the devices supported by this VI.

Route Signal

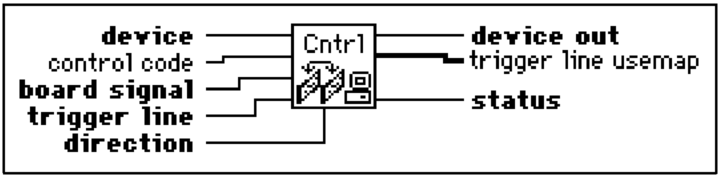
Use this VI to route an internal signal to the specified I/O connector or RTSI bus line, or to enable clock sharing through the RTSI bus clock line.

☞ **Note:** ***This VI is supported by E Series and 54XX Series devices only.***



RTSI Control

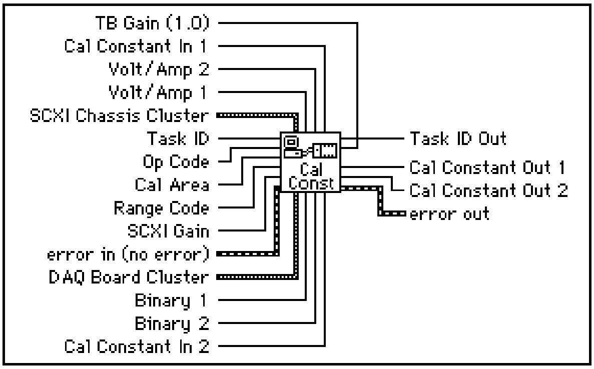
Connects or disconnects trigger and timing signals between DAQ devices along the Real-Time System Integration (RTSI) bus.



This VI is not supported for E Series devices. For E Series devices, multiple RTSI connections can be set directly in the analog input, analog output, and counter VIs and used along with the Route Signal VI. Other RTSI connections must be made using the Route Signal VI.

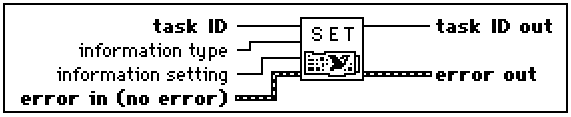
SCXI Cal Constants

Calculates calibration constants for the given channel and range or gain using measured voltage/binary pairs. You can use this VI with any SCXI module.



Set DAQ Device Information

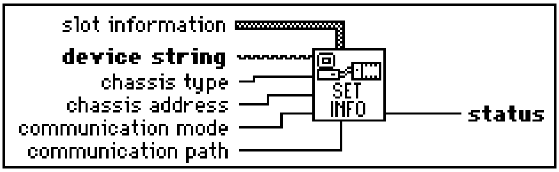
Sets the data transfer mode for different types of operations.



Refer to Appendix A, *DAQ Hardware Capabilities*, for the transfer methods available with your DAQ device.

Set SCXI Information

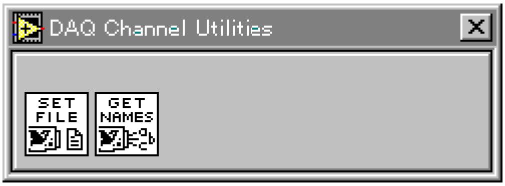
Sets the SCXI chassis configuration information.



Use this VI to override the configuration already set with the configuration utility You can use this VI *instead* of using the configuration utility to enter the chassis configuration information. If you do not use this VI, the first VI that accesses an SCXI chassis automatically tries to load information from the configuration file.

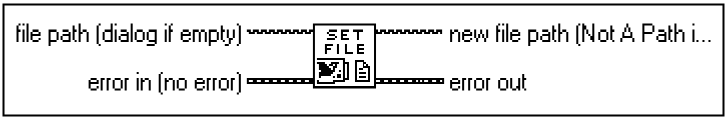
Channel Configuration VIs

The following illustration shows the Channel Configurations VIs palette.



Set DAQ Configuration File (Windows)

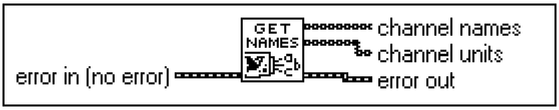
Sets the default DAQ Configuration file, which the NI-DAQ driver uses.



Note: ***This VI is specific to computers running Windows with NI-DAQ 5.0 or later. LabVIEW returns an UnsupportedError message if you attempt to run this VI on computers not running Windows.***

Get DAQ Channel Names (Windows)

Returns the an array of all the channel names in the default configuration file. A corresponding array of the channels' configured physical units is also returned.



Note: ***This VI is specific to computers running Windows with NI-DAQ 5.0 or later. LabVIEW returns an UnsupportedError message if you attempt to run this VI on computers not running Windows.***

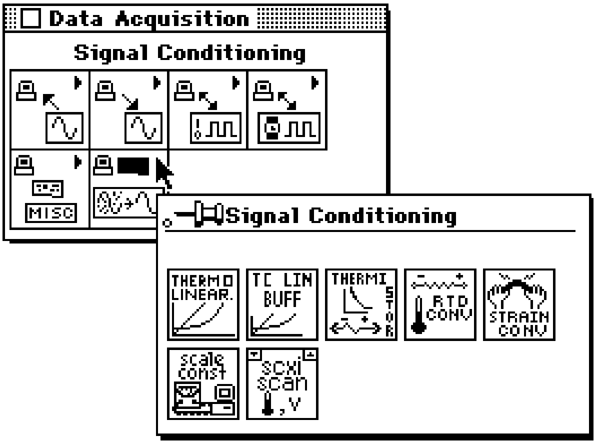
Chapter 29

Signal Conditioning VIs

This chapter describes the data acquisition Signal Conditioning VIs, which you use to convert analog input voltages read from resistance temperature detectors (RTDs), strain gauges, or thermocouples into units of strain or temperature.

You can edit the conversion formulas used in these VIs or replace them with your own to meet the specific accuracy requirements of your application. If you edit or replace the formulas, you should save the new VI in one of your own directories or folders outside of `vi.lib`.

You can access the Signal Conditioning VIs by choosing **Functions»Data Acquisition»Signal Conditioning**, as shown below.

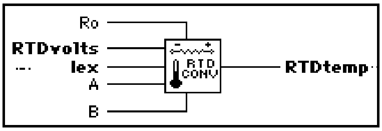


Signal Conditioning VI Descriptions

The following Signal Conditioning VIs are available.

Convert RTD Reading

Converts a voltage you read from an RTD into temperature in Celsius.



This VI first finds the RTD resistance by dividing **RTDVolts** by **Iex**. The VI then converts the resistance to temperature using the following solution to the Callendar Van-Dusen equation for RTDs:

$$Rt = Ro[1 + At + Bt^2 + C(t-100)t^3]$$

For temperatures above 0° C, the C coefficient is 0, and the preceding equation reduces to a quadratic equation for which the algorithm implemented in the VI gives the appropriate root. So, this conversion VI is accurate only for temperatures above 0° C.

Your RTD documentation should give you **Ro** and the **A** and **B** coefficients for the Callendar Van-Dusen equation. The most common RTDs are 100-Ω platinum RTDs that either follow the European temperature curve (DIN 43760) or the American curve. The following table gives the values for **A** and **B** for the European and American curves.

European Curve (DIN 43760)	American Curve
A = 3.90802e–03 **B** = –5.80195e–07 (α = 0.00385; ∂ = 1.492)	**A** = 3.9784e–03 **B** = –5.8408e–07 (α = 0.00392; ∂ = 1.492)

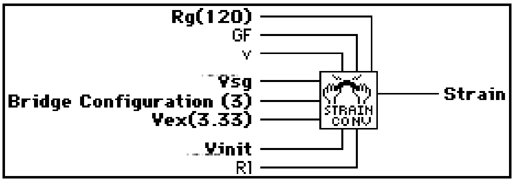
Some RTD documentation gives values for α and ∂, from which you can calculate **A** and **B** using the following equations:

$$\mathbf{A} = \alpha(1 + \partial/100)$$

$$\mathbf{B} = -\alpha\partial/100^2$$

Convert Strain Gauge Reading

Converts a voltage you read from a strain gauge to units of strain.



The conversion formula the VI uses is based solely on the bridge configuration. Figures 29-1 through 29-3 show the seven bridge configurations you can use and the corresponding formulas. For all bridge configurations, the VI uses the following formula to obtain **Vr**:

Vr = (**Vsg** – **Vinit**) / **Vex**

In the circuit diagrams, VOUT is the voltage you measure and pass to the conversion VI as the **Vsg** parameter. In the quarter-bridge and half-bridge configurations, R1 and R2 are dummy resistors that are not directly incorporated into the conversion formula. The SCXI-1121 and SCXI-1122 modules provide R1 and R2 for a bridge-completion network, if needed.

Refer to your *Getting Started with SCXI* manual for more information on bridge-completion networks and voltage excitation.

Figures 29-1 through 29-3 illustrate the bridge-completion networks available.

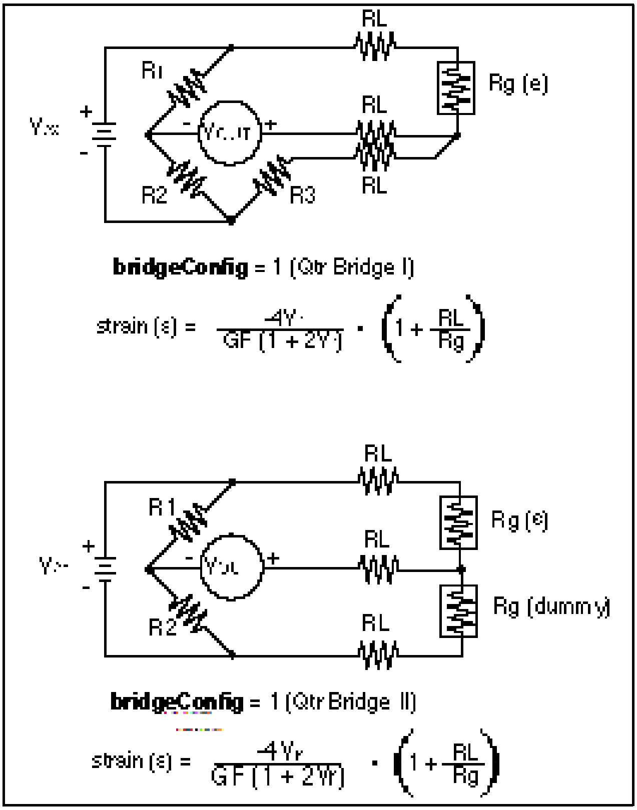

RL
R1
Rg (ε)
Vex
+
-
Vout
RL
R2
R3
RL
bridgeConfig = 1 (Qtr Bridge I)
strain (ε) = -4Vr / GF (1 + 2Vr) · (1 + RL/Rg)
RL
R1
Rg (ε)
Vex
+
-
Vout
RL
Rg (dummy)
R2
RL
bridgeConfig = 1 (Qtr Bridge II)
strain (ε) = -4Vr / GF (1 + 2Vr) · (1 + RL/Rg)

Figure 29-1. Strain Gauge Bridge Completion Networks (Quarter-Bridge Configuration)

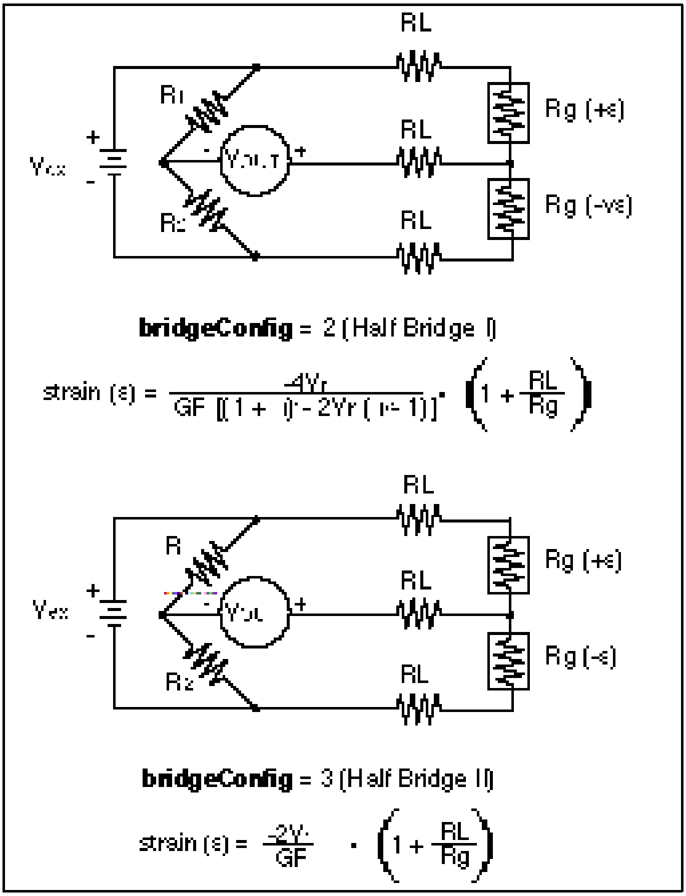

RL
R1
RL
Rg (+ε)
+
Vex
-
Vout
+
RL
Rg (-νε)
R2
bridgeConfig = 2 (Half Bridge I)
strain (ε) = -4Vr / GF [(1 + ν) - 2Vr (ν - 1)] · (1 + RL/Rg)
RL
R
RL
Rg (+ε)
+
Vex
-
Vout
+
RL
Rg (-ε)
R2
bridgeConfig = 3 (Half Bridge II)
strain (ε) = -2Vr/GF · (1 + RL/Rg)

Figure 29-2. Strain Gauge Bridge Completion Networks (Half-Bridge Configuration)

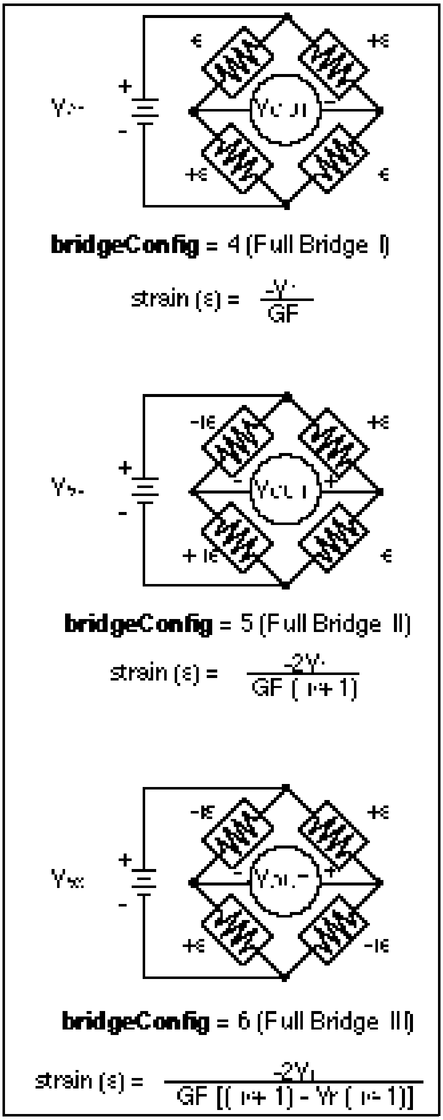
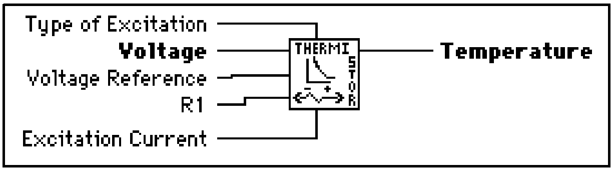

ε
+ε
Vex
+
-
Vout
+ε
ε
bridgeConfig = 4 (Full Bridge I)
strain (ε) = -Vr / GF
-Iε
+ε
Vex
+
-
-
Vout
+
+ Iε
ε
bridgeConfig = 5 (Full Bridge II)
strain (ε) = -2Vr / GF (ν+ 1)
-Iε
+ε
Vex
+
-
-
Vout
+
+ε
-Iε
bridgeConfig = 6 (Full Bridge III)
strain (ε) = -2Vr / GF [(ν+ 1) - Vr (ν- 1)]

Figure 29-3. Strain Gauge Bridge Completion Networks (Full-Bridge Configuration)

Convert Thermistor Reading

Converts a thermistor voltage into temperature. This VI has two different modes of operation for voltage-excited and current-excited thermistors.



This VI has two modes of operation for use with different types of thermistor circuits. Figure 29-4 shows how the thermistor can be connected to a voltage reference. This is the setup used in the SCXI-1303, SCXI-1322, SCXI-1327, and SCXI-1328 terminal blocks, which use an onboard thermistor for cold-junction compensation.

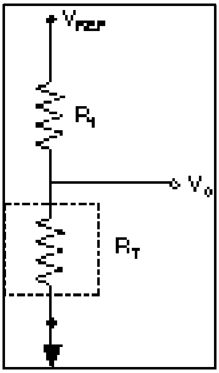


Figure 29-4. Circuit Diagram of a Thermistor in a Voltage Divider

Figure 29-5 shows a circuit where the thermistor is excited by a constant current source. An example of this setup would be the use of the DAQPad-MIO-16XE-50, which provides a constant current output. The DAQPad-TB-52 has a thermistor for cold-junction sensing.

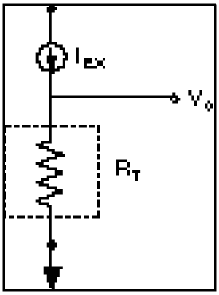


Figure 29-5. Circuit Diagram of a Thermistor with Current Excitation

If the thermistor is excited by voltage, the following shows equation relating the thermistor resistance, R_T, to the input values:

$$R_T = R_1\left(\frac{V_0}{V_{REF} - V_0}\right)$$

If the thermistor is current excited, the equation is

$$R_T = \frac{V_0}{I_{EX}}$$

The following equation is the standard formula the VI uses for converting a thermistor resistance to temperature:

$$T_K = \frac{1}{a + b(lnR_T) + c(lnR_T)^3}$$

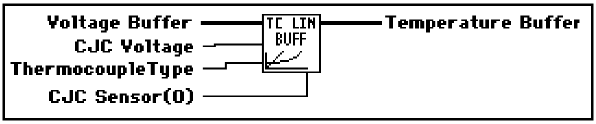
The values used by this VI for *a*, *b*, and *c* are given below. These values are correct for the thermistors provided on the SCXI and DAQPad-TB-52 terminal blocks. If you are using a thermistor with different values for *a*, *b*, and *c* (refer to your thermistor data sheet), you can edit the VI diagram to use your own *a*, *b*, and *c* values.

a =	1.295361E–3
b =	2.343159E–4
c =	1.018703E–7

The VI produces a temperature in degrees Celsius. Therefore, $T_C = T_K - 273.15$.

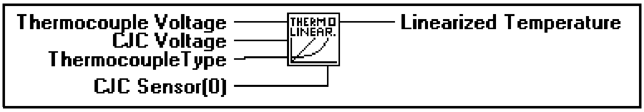
Convert Thermocouple Buffer

Converts a voltage buffer read from a thermocouple into a temperature buffer value in degrees Celsius.



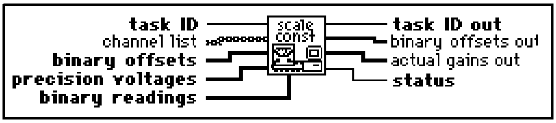
Convert Thermocouple Reading

Converts a voltage read from a thermocouple into a temperature value in degrees Celsius.



Scaling Constant Tuner

Adjusts the scaling constants, which LabVIEW uses to account for offset and non-ideal gain, to convert analog input binary data to voltage data.



To use this VI correctly, you must first take two analog input readings—a zero offset reading and a known-voltage reading.

The default binary offset for each channel in the group is 0. To determine the actual binary offset for a channel path, ground the channel inputs and take a binary reading, or take multiple binary readings and average them to get fractional LSBs of the offset.

If you use SCXI, ground the inputs of the SCXI channels to measure the offset of the entire signal path, including both the SCXI module and the DAQ device. The SCXI-1100, SCXI-1122, and SCXI-1141 modules have an internal switch you can use to ground the amplifier inputs without actually wiring the terminals to ground. To use this feature, type the special SCXI string CALGND in your SCXI channel string as described in the *Amplifier Offset* section of Chapter 19, *Common SCXI Applications*, in the *LabVIEW Data Acquisition Basics Manual*. Use intermediate or advanced analog input VIs to get binary data instead of voltage data.

☞ **Note:** ***If your device supports dithering, you should enable dither on your DAQ device when you take multiple readings and average them.***

LabVIEW assumes the DAQ devices gain settings and SCXI modules are ideal when it scales binary readings to voltage, unless you use this VI to determine actual gain values for the channels. Apply a known precision voltage to each channel and take a binary reading, or take multiple readings from each channel and compute an average binary reading for each channel. Your precision voltage should be about ten times as accurate as the resolution of your DAQ device to produce meaningful results. When you wire **binary readings**, **precision voltages**, and **binary offsets** to this VI, LabVIEW determines the actual gain using the following formula:

$$\text{actual gain} = \frac{\text{voltage resolution} * (\text{binary reading} - \text{binary offset})}{\text{precision voltage}}$$

In this formula, the **voltage resolution** value expressed in volts per LSB and is a value that varies depending on the DAQ device type, the polarity setting, and the input range setting. For example, the voltage resolution for a PCI-MIO-16E-1 device in bipolar mode with an input range of +5 to –5 V is 2.44 mV. The VI returns an array of the actual gain values that the VI stores for each channel.

☞ **Note:** ***When you take readings to determine the offset and actual gain, you should use the same input limits settings and clock rates that you use to measure your input signals.***

LabVIEW uses the following equation to scale binary readings to voltage:

$$\text{voltage} = \frac{\text{voltage resolution} * (\text{binary reading} - \text{binary offset})}{\text{gain}}$$

When you run the AI Group Config VI, it sets the attributes of all the channels in the group to their defaults, including the binary offset and gain values.

You can wire **channel list** if you want to adjust the scaling constants for a subset of the channels in the group. If you leave **channel list** unwired, the VI adjusts the scaling constants for all channels in the group. The VI uses the same method as the AI Hardware Config VI to apply values in the **binary offsets**, **precision voltages**, and **binary readings** input arrays That is, if you wired channel list first (at index 0) of the input arrays apply to the channels listed at index 0 of **channel list** if you wired **channel list**, or to the channels listed at index 0 of **channel list**. If you leave **channel list** unwired, the first values of the input arrays apply to the first channel in the group. The VI applies the values of each input array to **channel list** channels or the group in this manner until the VI exhausts the arrays. If channels in **channel list** or in the group remain unconfigured, the VI applies the final values in the arrays to all the remaining unconfigured channels.

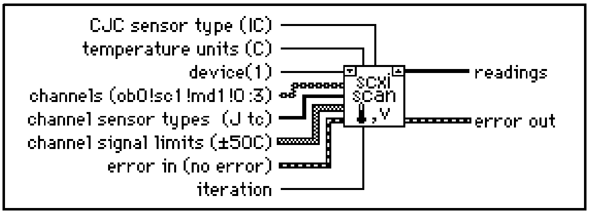
If you want to adjust only the channel offsets, and you want to assume the gain settings on the DAQ device and SCXI modules are ideal, wire only **binary offsets** and leave **precision voltages** and **binary readings** unwired.

You can also use this VI to retrieve the binary offset and actual gain values for all the channels in the group by wiring **taskID** only.

After you use this VI to adjust the scaling constants for a channel path, any analog input VIs that return voltage data use the adjusted constants for scaling. You can use the AI Group Config VI to reset the scaling constants for each channel in the group to their default values (zero offset and ideal gain).

SCXI Temperature Scan

This VI returns a single scan of temperature data from a list of SCXI channel. The SCXI Temperature Scan VI uses averaging to reduce 60 Hz and 50 Hz noise, performs thermocouple linearization, and performs offset compensation for the SCXI-1100 module.



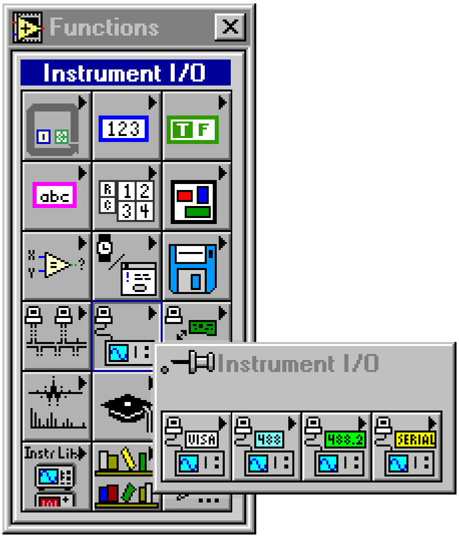
Introduction to LabVIEW Instrument Driver VIs

Chapter 30

This chapter includes an overview of LabVIEW instrument drivers and the GPIB, serial port, instrument driver template, and VISA VIs and functions. It also contains a history of the GPIB, and an explanation of GPIB improvements and standards. Descriptions of the VIs and functions comprise Chapter 31 through Chapter 37.

You can find the Instrument Driver VIs in the **Functions** palette from your block diagram in LabVIEW. The Instrument Driver VIs are located near the bottom of the **Functions** palette.

To access the **Instrument I/O** palette, choose **Functions»Instrument I/O**, as shown in the following illustration.



The Instrument I/O palette consists of the following subpalettes:

- VISA
- Traditional GPIB

- GPIB 488.2
- Serial

You can find helpful information about individual VIs online by using the LabVIEW Help window (**Help»Show Help**). When you place the cursor on a VI icon, the wiring diagram and parameter names for that VI appear in the Help window. You can also find information for front panel controls or indicators by placing the cursor over the control or indicator with the Help window open. For more information on the LabVIEW Help window, refer to the *Getting Help* section in Chapter 2, *Creating VIs*, of the *LabVIEW User Manual.*

In addition to the Help window, LabVIEW has more extensive online information available. To access this information, select **Help»Online Reference**. For most block diagram objects, you can select **Online Reference** from the object's pop-up menu to access the online description. For information on creating your own online reference files, see the *Creating Your Own Help Files* section in Chapter 25, *Managing Your Applications* of the *LabVIEW User Manual*.

Instrument Drivers Overview

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.

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.

Instrument Driver Distribution

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

You can download the latest versions of the LabVIEW instrument drivers from one of the National Instruments bulletin boards and, if you have internet access, you can download the latest instrument driver files from the National Instrument File Transfer Protocol site. See the Bulletin Board Support and FTP Support sections of Appendix E, *Customer Communication*.

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-on-Demand, at (512) 418-1111 or by calling National Instruments.

Instrument Driver Template VIs

The LabVIEW instrument driver templates are the foundation for all LabVIEW instrument driver development. The templates have a simple, flexible structure and a common set of instrument driver VIs that you can use for driver development. The VIs 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.

Chapter 33, *Instrument Driver Template VIs*, provides more information on the Instrument Driver Template VIs.

Introduction to VISA Library

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 string 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.

For more information on VISA functions, see Chapter 34, *VISA Library Reference*.

Introduction to GPIB

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

History of the GPIB

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/s maximum data transfer rates. It was later accepted as IEEE Standard 488-1975 and has since evolved into ANSI/IEEE Standard 488.2-1987. The versatility of the system prompted the name General Purpose Interface Bus. For a basic description of the GPIB, see Appendix C, *Operation of the GPIB*.

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 488.2 specification.

The 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. Appendix C, *Operation of the GPIB*, for more information on Talkers, Listeners, and Controllers.

Compatible GPIB Hardware

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

LabVIEW for Windows 95 and Windows 95-Japanese

- AT-GPIB/TNT, AT-GPIB/TNT (PnP), AT-GPIB/TNT+ ² PCI-GPIB
- PCMCIA-GPIB, PCMCIA-GPIB+
- GPIB-ENET
- EISA-GPIB
- VXIpc Model 850
- NEC-GPIB/TNT, NEC-GPIB/TNT (PnP)
- GPIB-PCII/IIA
- PC/104-GPIB
- CPCI-GPIB
- GPIB-ENET
- PMC-GPIB

LabVIEW for Windows NT

- AT-GPIB, AT-GPIB/TNT
- PCMCIA-GPIB
- PCI-GPIB
- VXIpc Model 850
- GPIB-ENET

LabVIEW for Windows 3.1

- AT-GPIB, AT-GPIB/TNT, AT-GPIB/TNT (PnP), AT-GPIB/TNT+ PCI-GPIB
- PCMCIA-GPIB, PCMCIA-GPIB+
- GPIB-ENET
- EISA-GPIB
- VXIpc Model 850
- NEC-GPIB/TNT (Japanese), NEC-GPIB/TNT (PnP) (Japanese) ² GPIB-PCII/IIA
- GPIB-232CT-A

- GPIB-485CT-A
- GPIB-1284CT
- PCII/IIA
- STD-GPIB
- EXM-GPIB
- MC-GPIB

LabVIEW for Mac OS

- PCI-GPIB
- NB-GPIB/TNT, NB-GPIB-P/TNT
- PCMCIA-GPIB
- LC-GPIB
- GPIB-ENET
- GPIB-232CT-A
- GPIB-SCSI-A
- PC/104-GPIB
- NB-DMA2800 (Traditional GPIB VI's only)

LabVIEW for HP-UX

- GPIB-ENET
- EISA-GPIB
- AT-GPIB/TNT

LabVIEW for Sun (Solaris)

- GPIB-ENET
- GPIB-SCSI-A
- SB-GPIB/TNT

LabVIEW for Concurrent PowerMAX

- GPIB-1014
- GPIB-1014D
- GPIB-1014P
- GPIB-1014DP

LabVIEW Traditional GPIB Functions

The traditional GPIB functions are compatible with all the GPIB boards listed in the *Compatible GPIB Hardware* section of this chapter.

These traditional GPIB functions are compatible with both IEEE 488 and IEEE 488.2 devices and are suffcient for most applications. For more complex applications, such as using several devices and more than one GPIB interface, you can use the GPIB IEE 488.2 functions.

For more information on the LabVIEW Traditional GPIB functions, see Chapter 35, *Traditional GPIB Functions*.

GPIB 488.2 Functions

Using 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.

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.

For more information on Single-Device Functions, see Chapter 36, *GPIB 488.2 Functions*.

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.

For more information on Multiple Device Functions, see Chapter 36, *GPIB 488.2 Functions*.

Bus Management Functions

The bus management functions perform system-wide functions or report system-wide status.

For more information on Bus Management functions, see Chapter 36, *GPIB 488.2 Functions*.

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.

For more information on Low-Level functions, see Chapter 36, *GPIB 488.2 Functions*.

General Functions

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

For more information on General functions, see Chapter 36, *GPIB 488.2 Functions*.

Serial Port VI Overview

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

For more information on serial port functions, see Chapter 37, *Serial Port VIs*.

Chapter 31

LabVIEW Instrument Driver Models

This chapter contains an overview of the LabVIEW instrument driver external interface model and the LabVIEW Instrument Driver Internal Design Model.

The following 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.

LabVIEW Instrument Driver External Interface Model

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

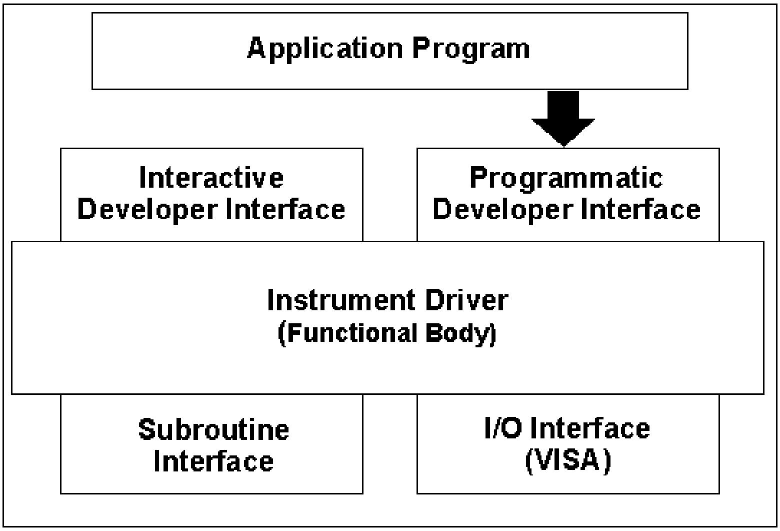

Application Program
Interactive Developer Interface
Programmatic Developer Interface
Instrument Driver (Functional Body)
Subroutine Interface
I/O Interface (VISA)

Figure 31-1. General Model of Instrument Drivers in LabVIEW

Functional Body

The *functional body* is the actual code for the instrument driver. Refer to the *LabVIEW Instrument Driver Internal Design Model* section of this chapter, for more information.

The most successful instrument driver products historically have been developed by using a standard programming language for the functional body. This is the approach LabVIEW instrument drivers take. 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 view instrument driver source code easily 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 panel, controls and indicators graphically represent the inputs and outputs of the VI. With the LabVIEW front panel, users can operate individual instrument driver VIs interactively 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 assemble test systems easily using 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 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, and other types of instruments. Refer to Chapter 34, *VISA Library Reference*, for further information.

Subroutine Interface

Because you write LabVIEW instrument drivers in standard LabVIEW graphical code, an instrument driver has 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 might 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 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.

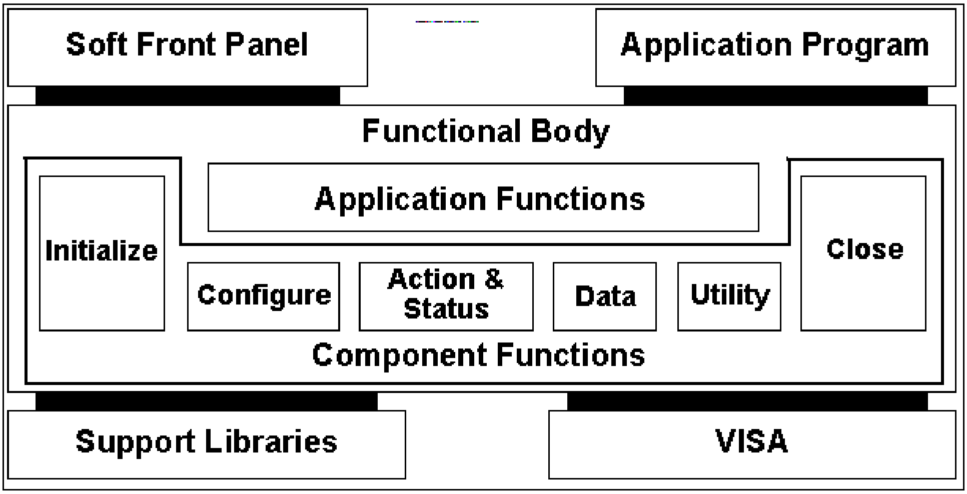


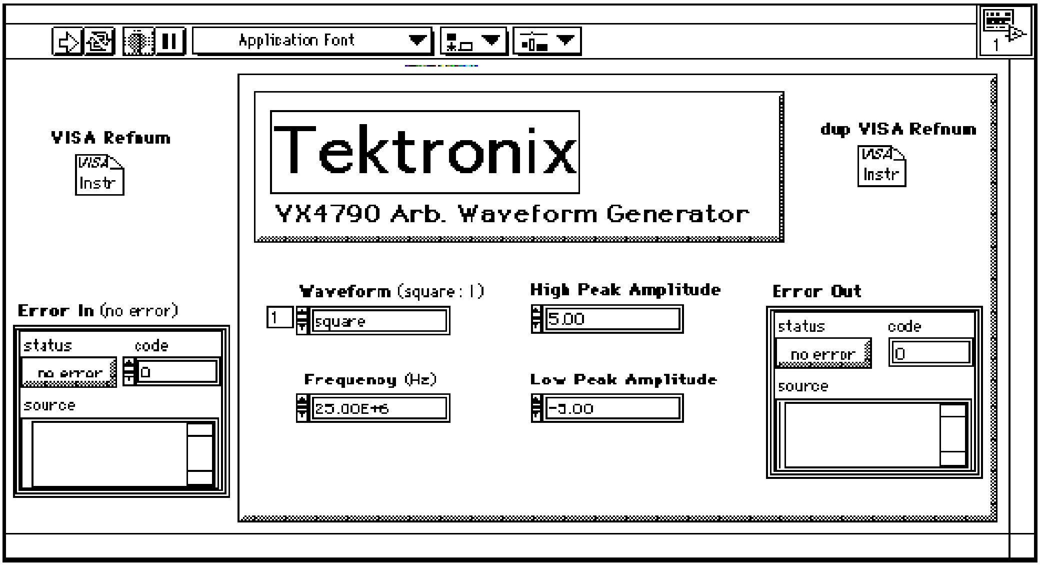
Figure 31-2. LabVIEW Instrument Driver Internal Design Model

The functional body of a LabVIEW instrument driver consists of two main categories of VIs. The first category is a collection of *component VIs*, which are individual software modules that each control a specific type of instrument function. The second category is a collection of higher-level *application VIs* that illustrate 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 beginning, configure multiple instruments, and then trigger several instruments simultaneously. As another example, you can initialize and configure an instrument once, and then trigger and read from the instrument several times.

Instrument Driver Application VIs

The *application VIs* are at the highest level of the instrument driver hierarchy. 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 high-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.



Error In (no error)
status: no error
code: 0
source:

Waveform (square : 1): 1 square

Frequency (Hz): 25.00E+6

High Peak Amplitude: 5.00

Low Peak Amplitude: -5.00

Error Out
status: no error
code: 0
source:

Figure 31-3. Tek VX4790 Example VI

The application VIs are built from a low-level set of instrument driver *component 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.

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.

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.

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. *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 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 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 query, error query, and error message and may include other custom instrument driver VIs, such as calibration or storing and recalling setups.

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

Each of these categories, with the exception of 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 *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 Chapter 33, *Instrument Driver Template 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.

Figure 31-4 shows how the Tek VX4790 Example application VI diagram uses the instrument driver component VIs:

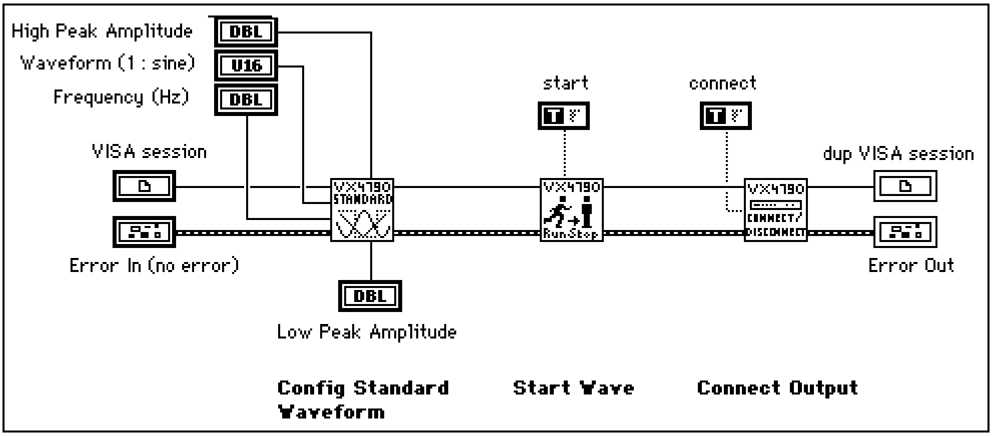


Figure 31-4. VIs in Tek VX4790 Example Diagram

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.

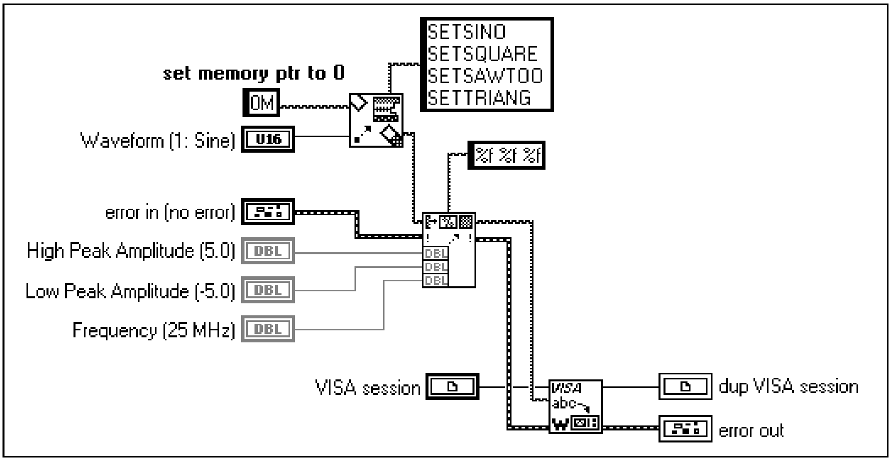


Figure 31-5. Tek VX4790 Config Std Wave Diagram

Error Reporting

LabVIEW instrument drivers use error clusters to report all errors. 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.

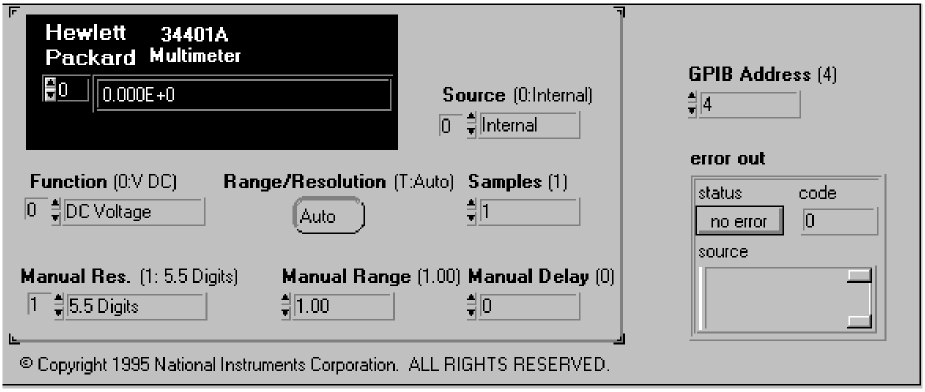
Another benefit of error input/output is that data dependency is added to VIs that are not otherwise data dependent.

Additional VIs Distributed with the Instrument Driver

In addition to the VIs described by the internal model, include a Getting Started VI and a VI Tree VI with your instrument driver files.

The Getting Started VI

The Getting Started VI allows the user to use the instrument without wiring a subVI on the block diagram. This is generally the first VI the end user runs to verify communication with the instrument. This VI generally consists of three sub-VIs: the initialize VI, an Application VI and the Close VI. The front panel of the Getting Started VI then resembles the application VI's front panel that it calls. Instead of having the user provide the VISA resource name, the user should only provide the GPIB address, VXI logical address or communications port. For example, instead of requiring the resource name "GPIB0::24", the Getting Started VI would require the user supply a GPIB address of "24." The front panel and block diagram of the Getting Started VI for the HP34401A are shown below.



Hewlett Packard 34401A Multimeter

0 0.000E+0

Source (0:Internal): 0 Internal

GPIB Address (4): 4

Function (0:V DC): 0 DC Voltage

Range/Resolution (T:Auto): Auto

Samples (1): 1

error out

status: no error

code: 0

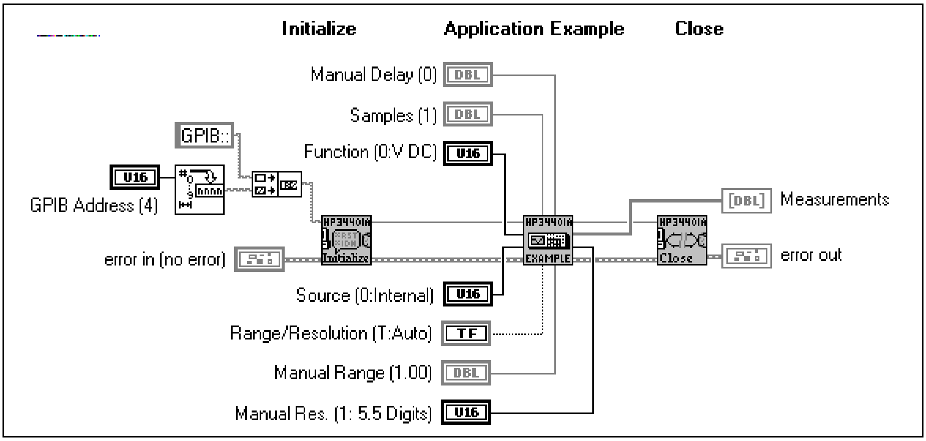
source:

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

Manual Range (1.00): 1.00

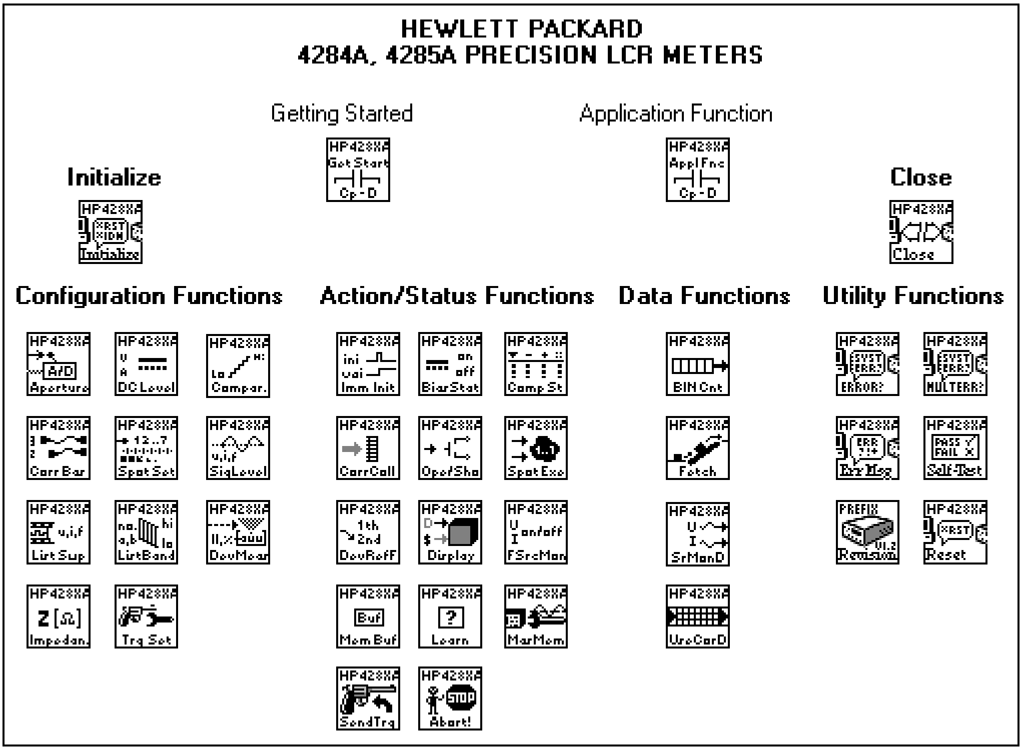
Manual Delay (0): 0

© Copyright 1995 National Instruments Corporation. ALL RIGHTS RESERVED.

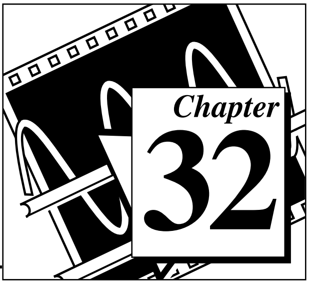


The VI Tree VI

In order for customers to view the entire instrument driver hierarchy at once, a VI Tree VI is required. This VI is a non-executable VI that is designed to show the functional structure of the VI. If an end user does not install the palette menu files for the instrument, the VI Tree is the only resource to understanding the structure. An example of a VI tree VI is shown below.



LabVIEW Instrument Driver Development



This chapter 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, this chapter focuses on the architectural guidelines of all drivers. With this information, driver developers can implement functionality unique to a particular instrument, and still organize, package and use all drivers in the same way.

Development Procedure

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

Designing the Instrument Driver Structure

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

When the number of programmable controls in an instrument increases, so does the need for modular instrument driver design 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 Chapter 33, *Instrument Driver Template VIs*.

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

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 query, 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 the VISA functions. 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/status, data, and utility.

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

Table 32-1. Instrument Driver Organization Example

VI Hierarchy	Type
Initialize VI	(Template)
Application VIs • Autosetup and Read Waveform • Rise-Time/Fall-Time Measurement	 (Developer Defined) (Developer Defined)

Table 32-1. Instrument Driver Organization Example (Continued)

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

Guidelines and Recommendations

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

  For example, a configure measurement VI would contain only the necessary controls to configure the instrument to take the measurement. It would not take the measurement or configure any other features. Other VIs 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.

Design Example

Deciding which parameters to include in an instrument driver VI is one of the greatest challenges facing the instrument driver developer. Fortunately, organizational information is often available in the instrument's 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.

Table 32-2. Command Summary from Tektronix VX4790

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 • Low-Pass filters	TKVX4790 Config Sample Frequency
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 Generator.

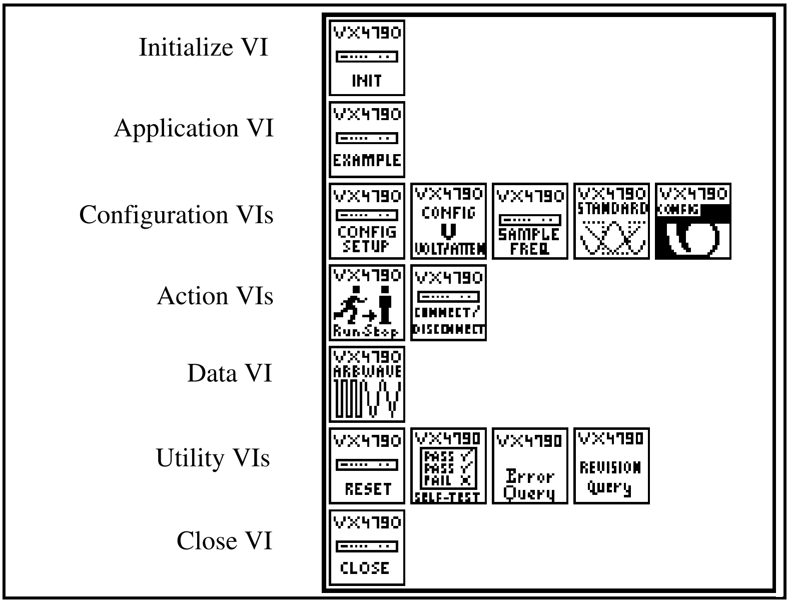


Figure 32-1. LabVIEW Instrument Driver VIs for the Tektronix VX4790

Modifying the Instrument Driver Templates

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

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

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

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

In addition to `CoreDrv.llb`, there is one more instrument driver template library, `CoreDr_U.llb`. This library can contain support VIs that the instrument driver uses internally, but which you do not intend the end user to call. Two examples of support files, PREFIX Utility Clean Up Initialize and PREFIX Utility Default Instrument Setup, are included in the `CoreDr_U.llb` file. If you intend the instrument driver to use these files, you should rename and modify them like those in `CoreDrv.llb`.

Adding Instrument Driver Component VI VIs

The final step in developing a LabVIEW instrument driver is to add the developer defined component VIs that define the functionality of the instrument driver and access the unique capabilities of your instrument. The VIs 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 `CoreDrv.llb`. Use the `PREFIX Message-Based` template VI for message-based operations. Use the `PREFIX Register-Based` template VI for register-based operations.
2. Edit the VI front panel. Create the controls and indicators for the VI.
3. Edit all control and indicator Help information. Edit the **Show VI Info...** description.
4. Edit the icon. Create an icon for each of the color modes of the icon: Black and White, 16-Color, and 256-Color.
5. Edit the connector pane. Select an appropriate connector pattern and wire all controls and indicators to the terminals.
6. Edit the block diagram. Program all operations necessary to carry out the functionality of the instrument driver VI.
7. Save the VI.
8. Test the instrument driver VI.
9. Repeat these steps for every instrument driver component VI and application VI that you define for your instrument.
10. Edit the instrument driver `.llb` by selecting **File»Edit VI Library...** from the menu. Edit the **Functions** and **Controls** names. Edit the arrangement of icons in the Functions and Controls palettes.

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.

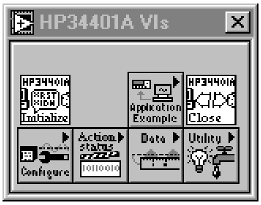
3. Wire the **VISA session** to every I/O VI.
4. Use the LabVIEW string VIs to assemble a command string based on the VI inputs.
5. Wire the command string to the `VISAWrite` function.
6. Use the `VISA Read` function to read the response if an instrument response is generated.
7. Use the string VIs to parse the response and wire it to the appropriate indicator terminals.

Modifying the Menu Files to Create Function Sub-Palettes

After you complete all the required VIs, component VIs, Application VIs and the Getting Started VI, organize them into subpalettes that the end user can access. This involves editing the template menu files as follows:

1. Copy the `CoreDrv` directory to another directory and rename the new directory `PREFIX`. This directory should be a subdirectory of `Instr.lib`.
2. Relaunch LabVIEW so that the new template subpalettes appear in the function palette under instrument drivers.
3. Select **Edit Controls and Function Palettes...** from the **File** menu in LabVIEW.
4. Edit the instrument driver's palette icon and change the name to `PREFIX`.
5. Access the instrument driver's subpalette window to view the hierarchy of the driver. For each subpalette, insert the VIs which correspond to that category. You will need to replace the template files with the completed version.
6. Save your changes. Your menu files will now contain the added component VIs.

The resulting menu palettes should resemble the following subpalette:



Tips for Developing a LabVIEW Instrument Driver

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. Figure 32-2 below shows the incorrect mechanisms for terminating a While Loop.

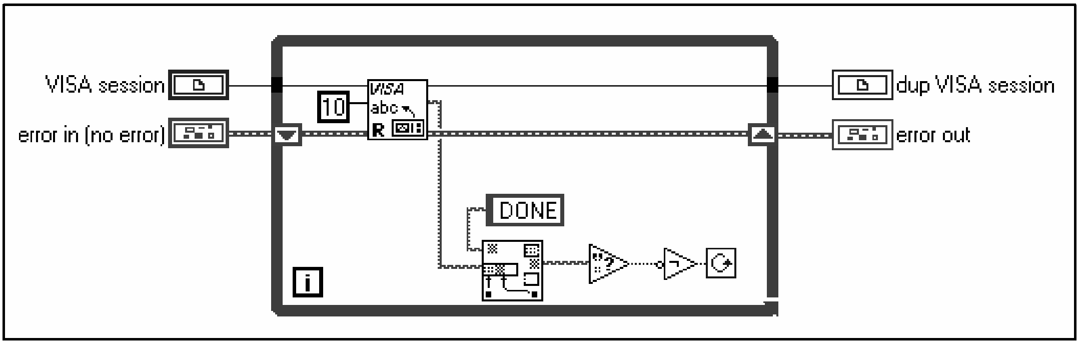


Figure 32-2. Incorrect Mechanism for Escaping from While Loop

Figure 32-3 below shows the correct mechanisms for terminating a While Loop

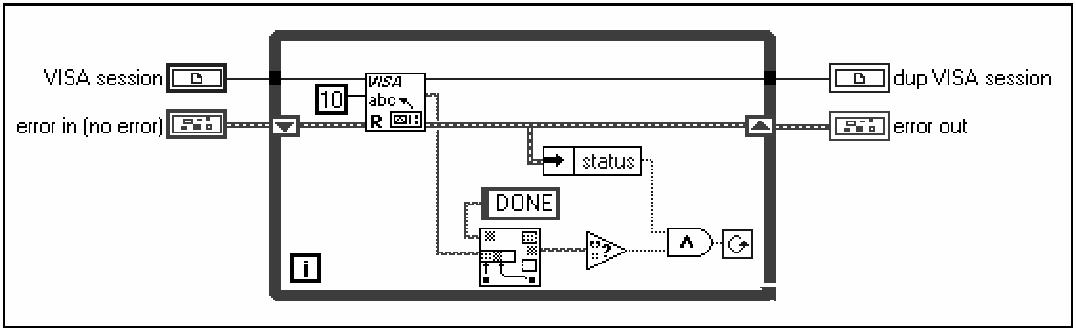
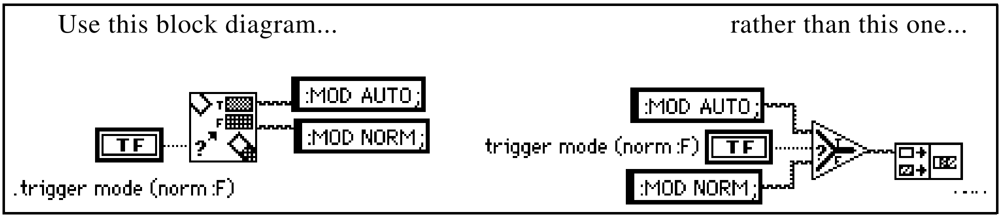


Figure 32-3. Correct Mechanism for Escaping from While Loop

Assembling Command Strings

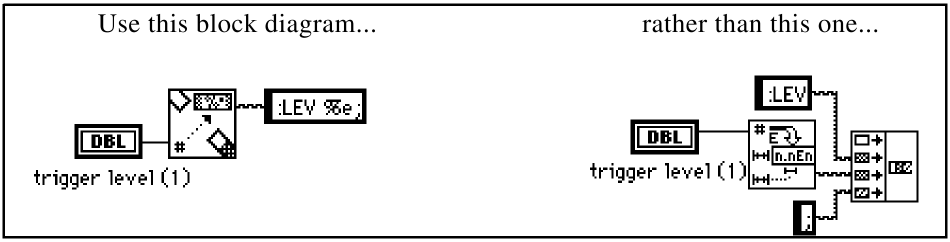
After you develop your front panel, the next step is to create the block diagram which performs the function required by the VI. Each type of front panel control has a corresponding block diagram string VI that simplifies the task of building command strings.

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.

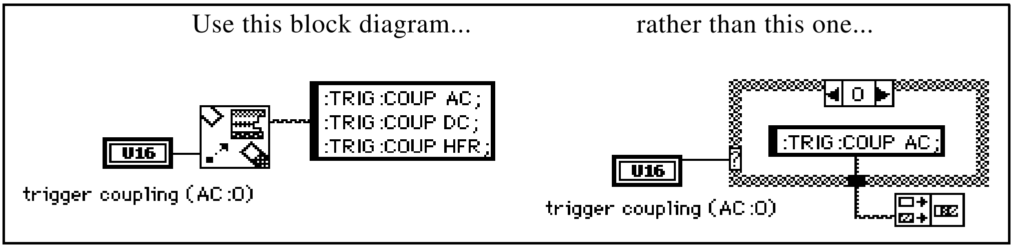


You can use Format & Append to format and concatenate simple numeric values. This procedure is easier than using one of the To

Decimal or To Exponential type conversion VIs in conjunction with Concatenate Strings.

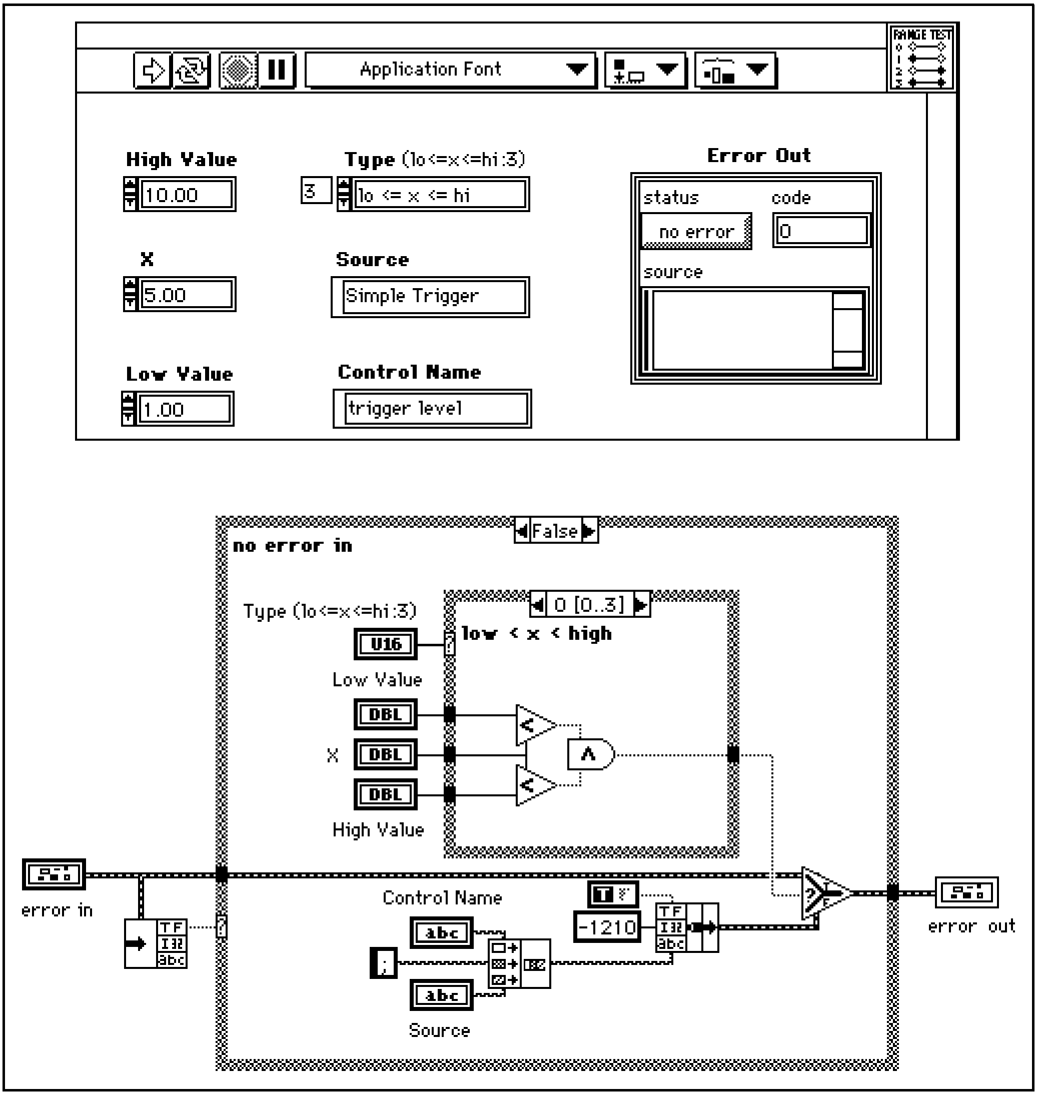


By using Select & Append you can select a string constant and concatenate it to another string in a single step. This procedure is easier than using Select and Concatenate Strings.



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 appropriate (see the *LabVIEW User Manual* for more information). 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 *Figure 31-3*, in Chapter 31, *LabVIEW Instrument Driver Models*. 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



Application Font

High Value: 10.00

Type (lo<=x<=hi:3): 3 lo <= x <= hi

X: 5.00

Source: Simple Trigger

Low Value: 1.00

Control Name: trigger level

Error Out
status: no error
code: 0
source:

Figure 32-4. Range Test VI (Front Panel and Block Diagram)

Programmatic range checking can easily double the size of your VI and add some execution speed penalties. Figure 32-5 and Figure 32-6 show the changes made to the Simple Trigger VI to programmatically check the ranges of the numeric inputs.

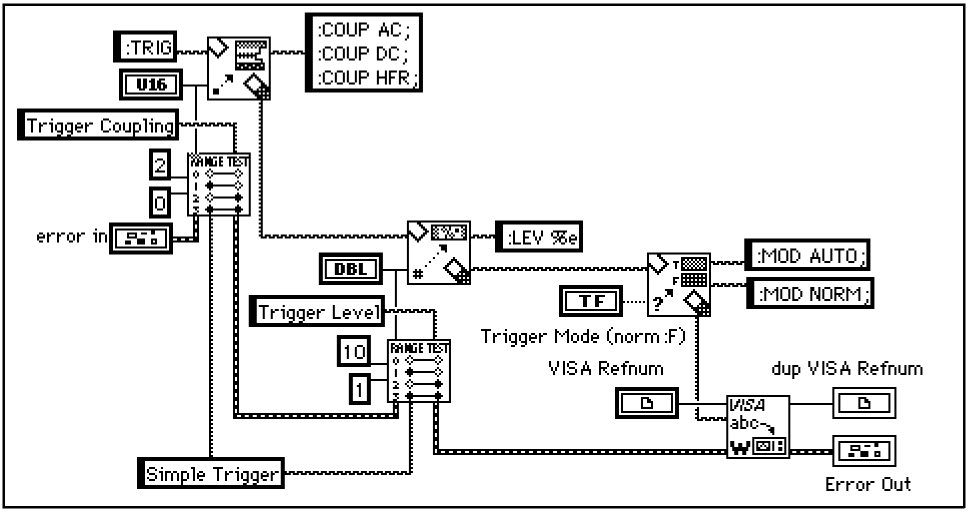


Figure 32-5. Simple Trigger VI with Programmatic Range Testing

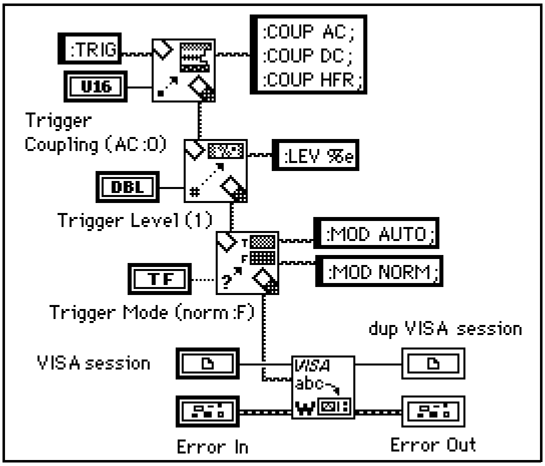


Figure 32-6. Simple Trigger VI without Programmatic Range Testing

Guidelines

Like the LabVIEW VI, the standard components of an instrument driver VI are the front panel, block diagram, and icon/connector pane. Special guidelines concerning these components, as well as error reporting and on-line help information, are described in the following sections.

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.

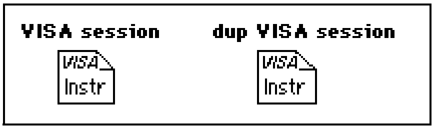
For example, you can represent any instrument command that has two options (such as TRIG:MODE:AUTO | NORMAL) on the front panel with a Boolean switch. In this case, label the switch **Trigger Mode** and add a free label showing the options: **auto** or **normal**. For commands that have a discrete number of options (such as TRIG:COUP:AC | DC | HFREJ), use a text ring or an enumerated type ring rather than a digital numeric because the ring control labels each numeric value with the command it represents. Any command requiring a numeric parameter whose value varies over a wide range and might be represented with a digital numeric.

☞ **Note:** ***You might prefer to use the enumerated type ring controls because selections for case structures are self-documenting when wired directly to a enumerated-type control or constant. Also, by using the "Create Constant" popup feature in LabVIEW, end users generate an enumerated type ring constant rather than a numeric constant.***

You can use Boolean, numeric, and text ring controls to represent most instrument commands on the front panels of your VIs. In addition, block diagram string functions specifically designed for use with these controls exist. These features can simplify string formatting and append instrument commands into command messages, as discussed in the *Assembling Command Strings* and *Block Diagram* sections of this chapter.

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`.

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 *LabVIEW Error Codes* manual for a description of the possible error codes.

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 out** cluster in the lower right corner. Place the **error in** cluster outside the panel's visible window because it has no interactive use 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 Instrument Driver VI front panels:

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

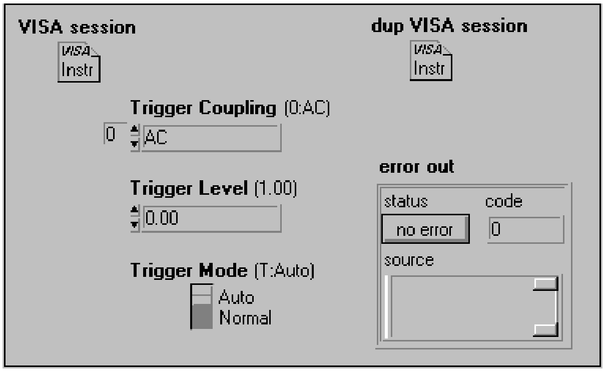
☞ **Note:** ***In most cases, all instrument driver controls are primary and require bold text. If you are finding yourself placing many secondary or auxiliary controls on panels, this may indicate the need to subdivide your VI into two or more VIs.***

- Capitalize initial letters in all words, except abbreviations or acronyms, which require caps (such as ID or GPIB) and **error in, error out** and **dup VISA session** labels.
- Place labels above the associated control or indicator and color the label background transparent.
- Enclose control default information in parentheses in the control name.

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

  For example, **Function** (0:DCV) would be an appropriate label for a function selector ring control whose default is DC voltage and item zero in the ring. The abel for a Boolean mode switch that defaults to true indicating automatic would be **Mode** (T:Auto). (Notice that the default information is in plain text)**.**
- Align and distribute the controls and indicators for a well balanced panel.

The following figures shows the simple trigger VI after modification to meet the style guidelines.



VISA session
Instr

dup VISA session
Instr

Trigger Coupling (0:AC)
0 AC

Trigger Level (1.00)
0.00

Trigger Mode (T:Auto)
Auto
Normal

error out
status: no error
code: 0
source

Figure 32-7. Simple Trigger VI Front Panel (See *Figure 32-8* for Diagram)

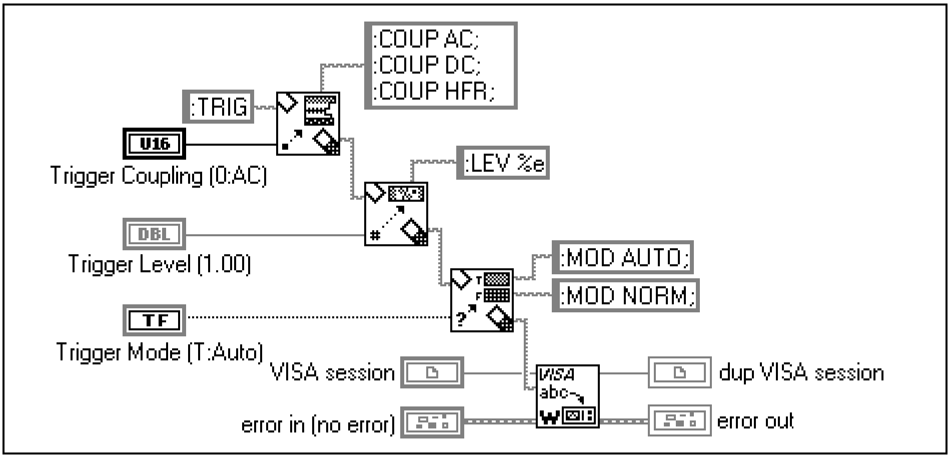


Figure 32-8. Simple Trigger Block Diagram

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. 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.***

Error Reporting

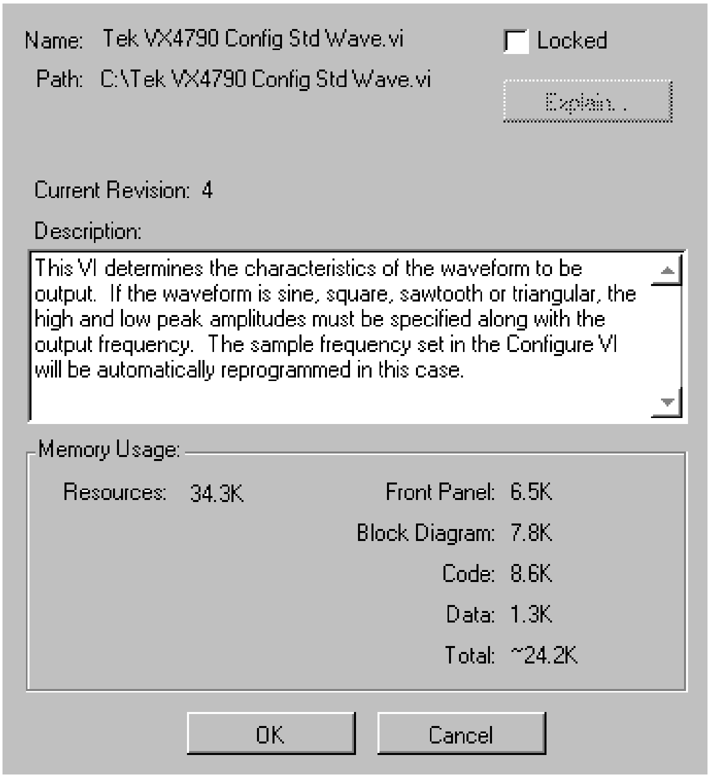
Refer to the document, *LabVIEW Error Codes,* for a list of error codes reserved for LabVIEW instrument drivers.

Online Help Information

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

VI Descriptions

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.



Name: Tek VX4790 Config Std Wave.vi

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

☐ Locked

Current Revision: 4

Description:

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

Memory Usage:

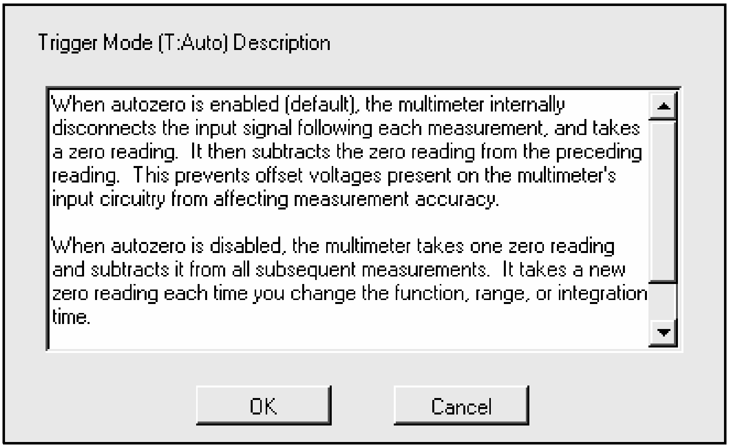
Resources:	34.3K	Front Panel:	6.5K
		Block Diagram:	7.8K
		Code:	8.6K
		Data:	1.3K
		Total:	~24.2K

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.



Trigger Mode (T:Auto) Description

When autozero is enabled (default), the multimeter internally disconnects the input signal following each measurement, and takes a zero reading. It then subtracts the zero reading from the preceding reading. This prevents offset voltages present on the multimeter's input circuitry from affecting measurement accuracy.

When autozero is disabled, the multimeter takes one zero reading and subtracts it from all subsequent measurements. It takes a new zero reading each time you change the function, range, or integration time.

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 controls.

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 described in this section. Use the following checklist to verify that your instrument driver complies with library standards:

I. Files and Documents you submit:

_____ A. `Prefix.zip` containing the instrument driver files.

_____ 1) `Prefix.llb`, your main instrument driver library. (e.g., `hp16500b.llb`, `fl45.llb`)

_____ 2) Palette menu files. (`dir.mnu, acstat.mnu, data.mnu, applic.mnu, util.mnu. config.mnu`)

_____ 3) (optional/recommended) `Prefix_u.llb`. (e.g., `hp1650_u.llb, fl45_u.llb`).

_____ 4) (optional) `Prefix.txt`.

_____ B. Manufacturer’s instrument manual or manual set.

_____ C. A completed checklist.

II. General Issues:

_____ A. The instrument driver must use VISA for all instrument I/O:

_____ B. All VIs are designed for programmatic use, so there are no pop-up VIs or dialog boxes, and no interactive inputs. All controls and indicators are wired to the connector pane.

_____ C. All VIs are multi-instance, so there are no uninitialized shift registers, and no global storage VIs unless specifically designed to work with multiple instruments simultaneously.

_____ D. All VIs are fully documented including Show VI Info and control descriptions.

_____ E. Driver follows the instrument internal and external driver model: The driver must include the following VIs: Initialize, Close, Getting Started, Application and VI Tree. In addition, all other VIs must be grouped into the following categories: Configure, Action/Status, Data, Utility, or support.

_____ F. All VIs use the error I/O clusters, **error in** and **error out**.

_____ G. The instrument driver contains the following required Utility functions: Revision Query, Self Test, Reset, Error Query (single and/or multiple), and Error Message.

_____ H. The required utility VIs return a VISA NSUP warning code if the instrument does not support the requested operation.

_____ I. The instrument driver uses **VISA session**, **dup VISA session**, **error in**, and **error out** to channel data flow, and force data dependency. Do not use sequences or case structures for this purpose because they slow execution speed and make it harder to debug the driver.

III. `Prefix.llb`:

_____ A. `Prefix.llb` contains all the instrument driver VIs that you want the end user to access directly.

_____ B. All VIs are saved with meaningful names including instrument prefix and description, and include only alpha-numeric characters (no special characters). Use Initial Capital Letter form (e.g., Fluke 45 Read Measurement). VIs that are of the same type should be named so that they start with a common name. For example, all configuration VIs should start with “Prefix Config.”

_____ C. The VI Tree is contained in `prefix.llb` and is named `Prefix VI Tree.vi`. The front panel of the VI contains a message instructing the users to “See the diagram for the VI Tree”. The diagram contains all of the driver's VIs that are designed for the user to access. These VIs are arranged by functional grouping, such as Getting Started, Application, Initialize, Configuration, Action/Status, Data, Utility, and Close.

_____ D. All instrument drivers have at least one Application VI. These VIs are programmatic examples that demonstrate how to use the instrument driver component VIs to perform a common task or tasks.

_____ E. All instrument drivers must have a Getting Started VI. This VI calls the Initialize VI, one or more application VIs, followed by the Close VI.

_____ F. Getting Started, Application and VI Tree VIs are given top-level status in the VI library.

IV. `Prefix_U.llb` (Recommended/Optional):

_____ A. `Prefix_U.llb` contains all the support VIs the end user should not access directly, but are used by the instrument driver.

V. Palette Menu Files:

_____ A. The function menu palettes are well organized and follow the format of the instrument driver template.

_____ B. Palette Menu files include `dir.mnu`, `applic.mnu`,`config.mnu`, `acstat.mnu`, `data.mnu`, and `util.mnu`.

VI. VI Front Panels

_____ A. Contains **VISA session**, **dup VISA session**, **error in** and **error out** controls/indicators.

_____ B. Front panel Show VI Info description is complete, informative, and contains any additional information that helps the end user successfully operate the instrument driver.

_____ C. Show VI Info for the Revision Query VI includes the following:

_____ 1) The Instrument Driver Revision Number

_____ 2) The Firmware Revision of the Instrument used when creating the instrument

_____ 3) The date the driver will be released on the next Instrument Driver CD (month/year)

_____ 4) The instrument manufacturer's name

_____ 5) The instrument model number

_____ 6) The instrument type (Digital Multi-Meter, Oscilloscope, Function Generator, etc.)

_____ 7) The instrument driver developer’s name

_____ D. (optional/recommended). The same information that is included in the revision query VI is included in the Show VI Info documentation of the VI Tree VI.

_____ E. VI History is updated with comments as needed.

_____ F. Controls and Indicators

_____ 1) All control and indicator descriptions are complete. This includes valid ranges, default values and items within a ring control.

_____ 2) Labels are placed at the upper left of controls and the background color of labels is transparent. Size to Text feature used.

_____ 3) Default Application Font is used and the initial letters of control names are capitalized. Use bold for primary controls and plain text for secondary controls. Use plain text to indicate default values.

_____ 4) Proper defaults are set for each control. Default values are included in the control name.

_____ 5) Proper data type and display format is used.

_____ 6) Enumerated text rings used instead of regular test rings, whenever possible.

_____ G. Align and distribute the controls for an appealing panel layout. Do not overlap controls. Set Panel Order so that users can tab through the controls in a logical sequence.

_____ H. Use color sparingly or use standard gray. If color is desired, use only the 16 basic colors.

VII. Icon/Connector Pane/VI Setup

_____ A. Create meaningful icons for all VIs.

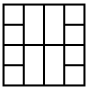
_____ 1) Place the instrument Prefix at the top of the icon. Place a text description at the bottom of the icon.

_____ 2) Try to keep a common theme for all VIs of a particular driver or group within a driver.

_____ 3) Black and white icons are required, 16 and 256 color icons are recommended/optional.

_____ B. Select an appropriate connector pane.

_____ 1) For ease of wiring it is recommended that the following connector pane is used, whenever possible:



_____ 2) Whenever possible, input terminals should be kept to the left and top while outputs are on the bottom and right.

_____ 3) VISA session must be assigned to the upper left terminal and dup VISA session is assigned to the upper right terminal. Similarly, error in and error out are assigned to the lower left and lower right terminals, respectively.

_____ 4) If future modifications are expected, a connector pane with extra unused terminals is acceptable.

_____ C. Use caution when using VI Setup options. Do not select options to make the panel automatically shown or run.

VIII. Block Diagram:

_____ A. Use bold text labels with 14 point application font to describe each case or sequence frame. These descriptions should be left-justified with the background colored transparent.

_____ B. Use plain text labels for controls/indicators with the default application font. For control terminals, place labels below or on the left. For indicator terminals, palce the labels below or on the right. If you place labels to the left of the terminal, make them right justified, otherwise use left-justification. Make label backgrounds transparent.

_____ C. Do not crowd the diagram. Do not cover wires with labels, objects, or structures.

_____ D. When possible, try to wire the VIs the way that they appear in the LabVIEW Help Window.

_____ E. Try to lay out the diagram with a left-right, top-down data flow.

_____ F. For functions and VIs that are chained together using the VISA sessions and error clusters, try to align the wires between sequences to be on the same horizontal level.

_____ G. Try to align and distribute terminals, VIs and functions within your block diagram to give it a well-balanced look. Eliminate unnecessary bends in the wires.

_____ H. Use proper error I/O wiring techniques. Use the correct error codes for error reporting.

_____ I. Save diagrams with the first or most important frames and cases visible. Place bold-text descriptive free labels in each case and frame.

_____ J. Avoid using sequence structures because they slow execution of your VI and make it harder to understand the diagram.

_____ K. Avoid using the Concatenate Strings function when another string function is more appropriate. Use other string handling functions such as Pick Line & Append, Select & Append and Format into String.

Chapter 33

Instrument Driver Template VIs

This chapter describes the Instrument Driver Template VIs. These VIs are located in `examples\instr\insttmpl.llb`.

Introduction to Instrument Driver Template VIs

The LabVIEW instrument driver templates are the foundation for all LabVIEW instrument driver development. The templates 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 contain the following 11 predefined template component VIs:

- PREFIX Initialize
- PREFIX Initialize (VXI, Reg-based)
- PREFIX Close
- PREFIX Reset
- PREFIX Self Test
- PREFIX Error Query
- PREFIX Error Query (Multiple)
- PREFIX Error Message
- PREFIX Revision Query
- PREFIX Message-Based Template
- PREFIX Register-Based Template

The templates contain the following support VIs:

- PREFIX Revision Query
- PREFIX Message-Based Template

They also contain the following VI Example Tree:

- PREFIX Message-Based Template

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.

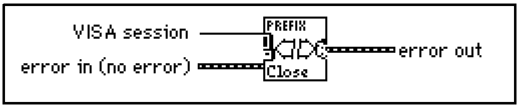
Instrument Driver Template VI Descriptions

The following Instrument Driver Template VIs are available.

☞ **Note:** ***To develop your own Instrument Driver VI, follow the instructions on the front panel of the Template VI.***

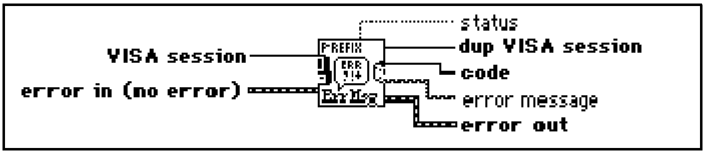
PREFIX Close

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..



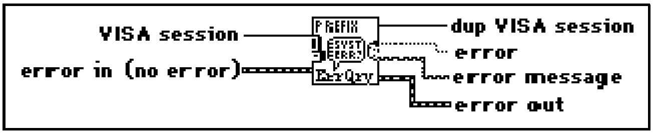
PREFIX Error Message

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.



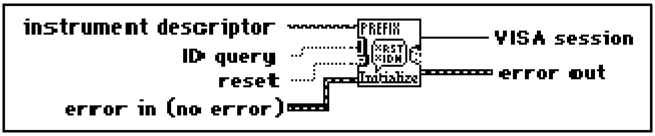
PREFIX Error Query, Error Query (Multiple) and Error Message

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.



PREFIX Initialize and PREFIX Initialize (VXI, Reg-based)

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 Initialize VI is a template for message-based instruments while Initialize (VXI, Reg-based) is for register-based instruments.



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.

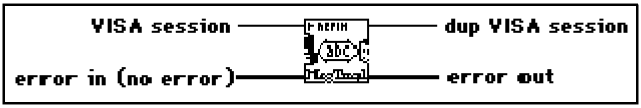
The following table shows the default values for optional parameters:

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

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.

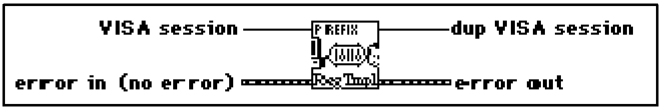
PREFIX Message-Based Template and Register-Based Template

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.



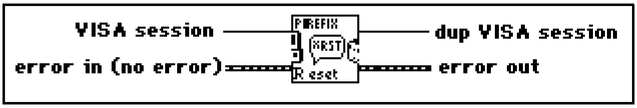
PREFIX Register-Based Template

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



PREFIX Reset

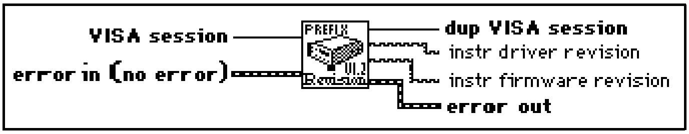
All LabVIEW instrument drivers have a *Reset VI* that places the instrument in a default state. The default state that the Reset VI places the instrument in should be documented in the help information for the Reset VI. In an IEEE 488.2 instrument, this VI sends the command string *RST to the instrument.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



PREFIX Revision Query

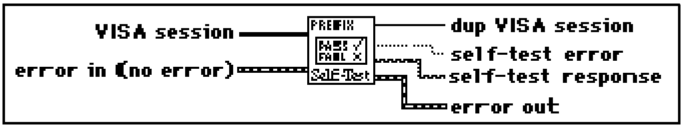
LabVIEW instrument drivers have a *Revision Query VI*. This VI outputs the following:

- 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`.)



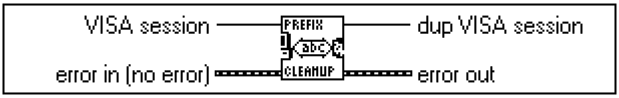
PREFIX Self-Test

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.



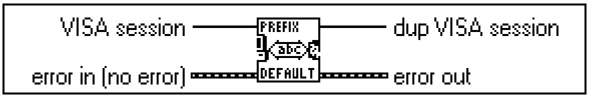
PREFIX Utility Clean UP Initialize

Closes an open VISA session in the event that there is an error during initialization. This VI should be called only from the Initialize VI.



PREFIX Utility Default Instrument Setup

Sends a default command string to the instrument whenever a new VISA session is opened, or the instrument is reset. Use this VI as a subVI for the Initialize and Reset VIs.



PREFIX VI Tree

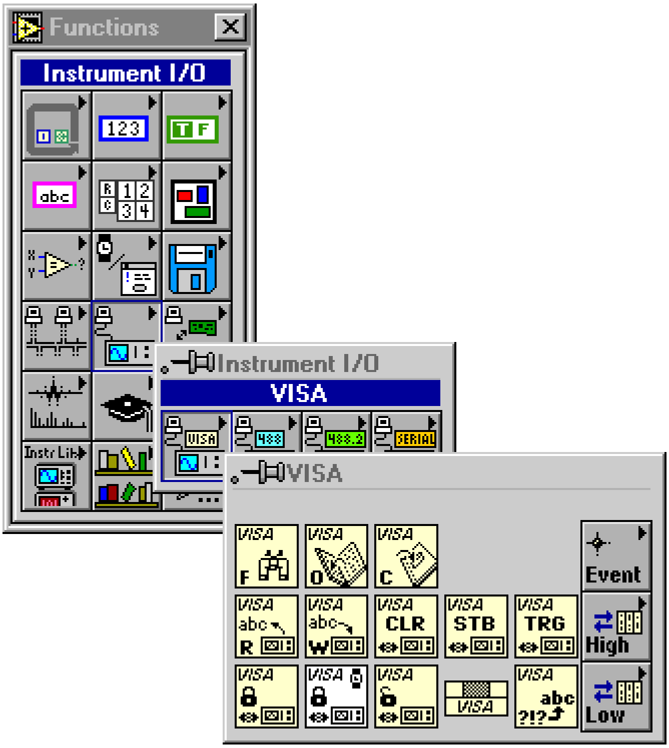
The VI Tree VI is a non-executable VI that shows the functional structure of the instrument driver. It contains the Getting Started VI, application VIs, and all of the component VIs.

Chapter 34

VISA Library Reference

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

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



The Visa palette includes the following subpalettes:

- Event Handling Functions
- High-Level Event Access
- Low-Level Registry Access

Operations

This section describes the VISA Library Reference operations.

VISA Library Reference Parameters

Most of the VISA Library Operations use the following parameters:

- 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
  - VXI/VME Interrupt Event
  - Resource Manager

Note: ***The Generic Event, Service Request Event, Trigger Event, VXI Signal Event, VXI/VME Interrupt Event, and Resource Manager 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*.

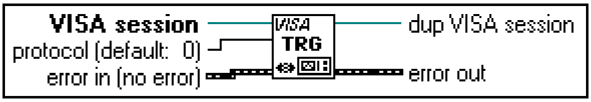
- **error in** and **error out** terminals comprise the error clusters in each VISA function. The error cluster contains three fields. The status field is a Boolean which is TRUE when an error occurs, FALSE when no error occurs. The **code field** will be a VISA error code value if an error occurs during a VISA function. Appendix D lists the VISA Reference Library error codes. 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.

VISA Operation Descriptions

These functions appear on the main VISA palette. The valid classes for these functions are: Instr (default), GPIB Instr, Serial Instr, VXI/GPIB-VXI RBD Instr, and VXI/GPIB-VXI MBD Instr.

VISA Assert Trigger

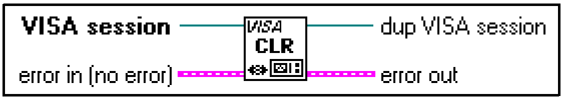
Asserts a software or hardware trigger, depending on the interface type.



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

VISA Clear

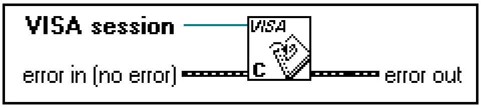
Performs an IEEE 488.1-style clear of the device. For VXI, this is the Word Serial Clear command; for GPIB systems, this is the Selected Device Clear command.



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

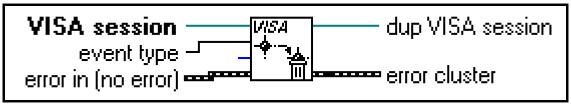
VISA Close

Closes a specified device session or event object. VISA Close accepts all available classes. For a listing of available classes, see the *VISA Operation Parameters* section earlier in this chapter.



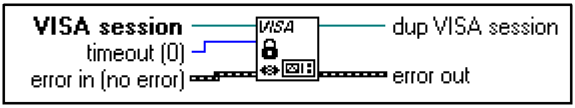
VISA Find Resource

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



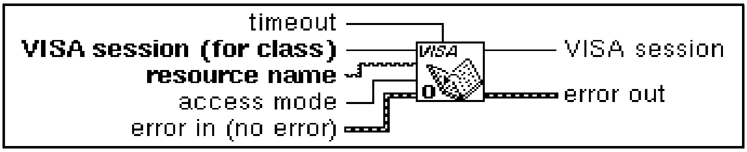
VISA Lock

Establishes exclusive access to the specified source.



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.



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]

Interface	Grammar
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

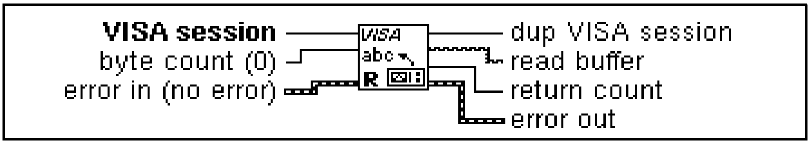
The following table shows examples of address strings.

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 `VISAClose` description earlier in this chapter.

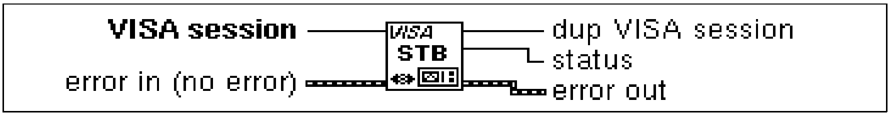
VISA Read

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



VISA Read STB

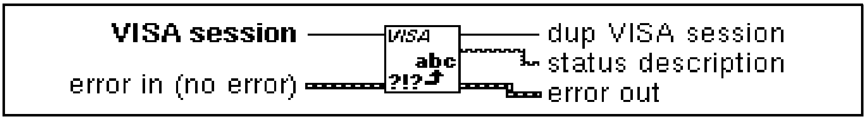
Reads a service request status from a message-based device. For example, on the IEEE 488.2 interface, the message is read by polling devices. For other types of interfaces, a message is sent in response to a service request to retrieve status information. If the status information is only one byte long,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



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

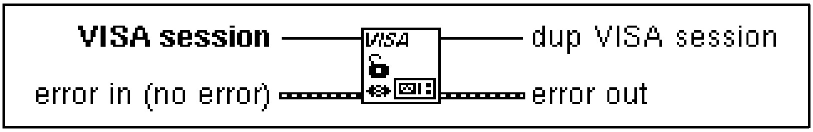
VISA Status Description

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



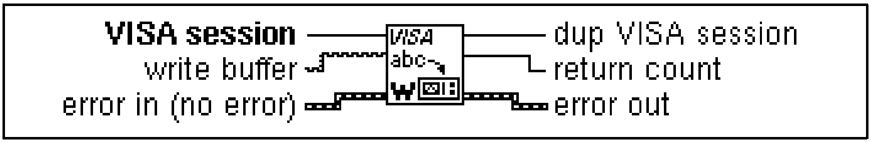
VISA Unlock

Relinquishes the lock previously obtained using the VISA Lock function.



VISA Write

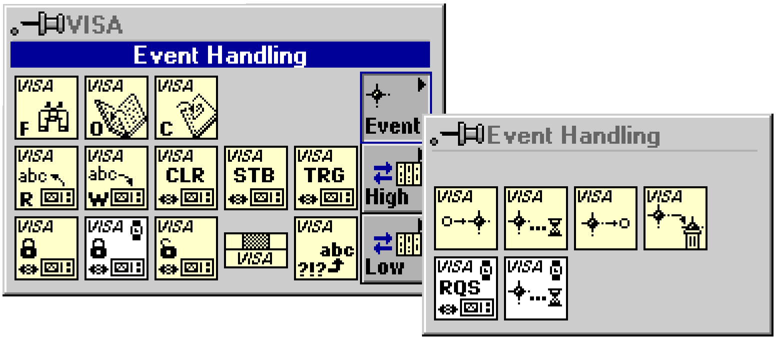
Writes data to the device. On UNIX platforms data is written synchronously; on all other platforms data is written asynchronously.



Event Handling Functions

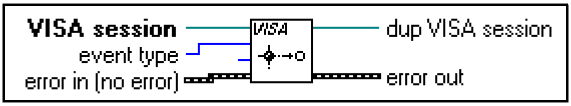
The following section describes the VISA Event Handling functions. Valid classes for these functions are: Instr (default), GPIB Instr, Serial Instr, VXI/GPIB-VXI RBD Instr, and VXI/GPIB-VXI MBD Instr.

You access the VISA Event Handling functions through the **VISA** palette, which you access by selecting **Functions»Instrument I/O»VISA.**



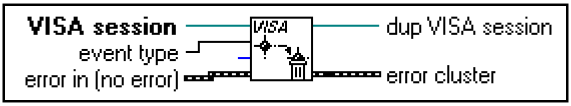
VISA Disable Event

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.



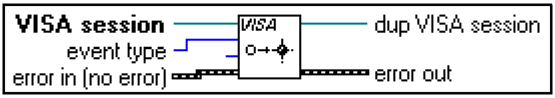
VISA Discard Events

Discards all pending occurrences of the specified event types and mechanisms from the specified session.



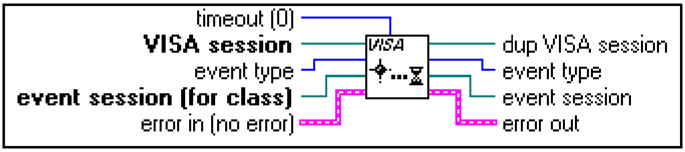
VISA Enable Event

Enables notification of a specified event.



VISA Wait On Event

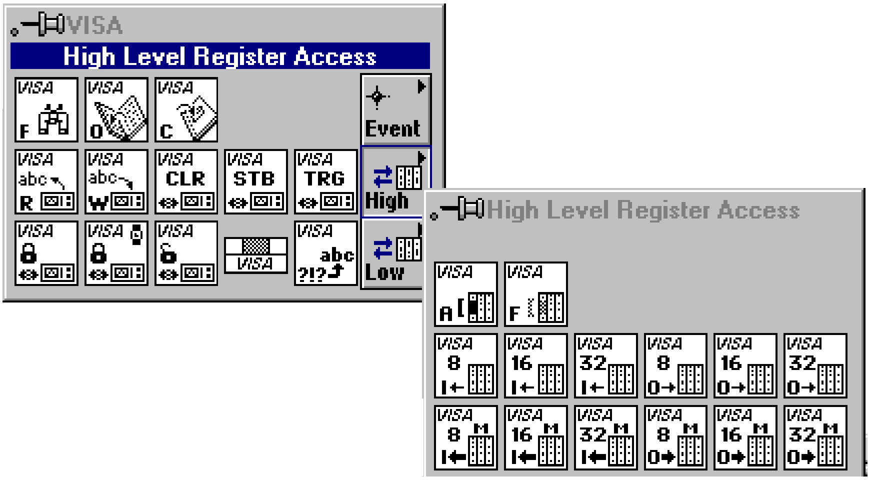
Suspends execution of a thread of application and waits for an event Event Type for a time period not to exceed that specified by timeout. Refer to individual event descriptions for context definitions. If the specified event type is All Events, the operation waits for any event that is enabled for the given session.



High Level Register Access Functions

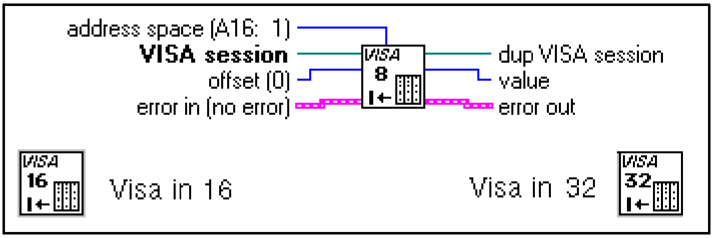
The following section describes the VISA High Level Register Access functions. Valid classes for these functions are: Instr (default), VXI/GPIB-VXI RBD Instr, and

VXI/GPIB-VXI MBD Instr. To access the VISA High Level Register Access functions, pop up on the High Level icon on the **VISA** palette.



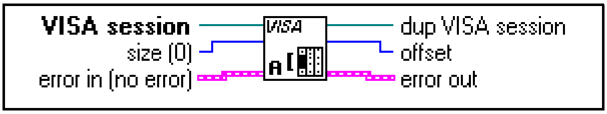
VISA In8 / In16 / In32

Reads in 8-bits, 16-bits, or 32-bits of data, respectively, from the specified memory space (assigned memory base + offset).



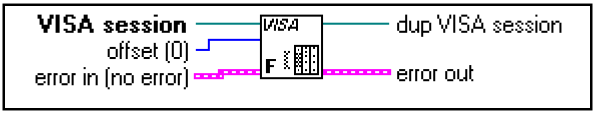
VISA Memory Allocation

Returns an offset into a device's region that has been allocated for use by the session. The memory can be allocated on either the device itself or on the computer's system memory.



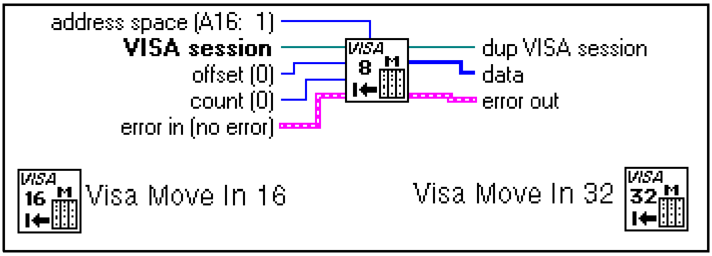
VISA Memory Free

Frees the memory previously allocated by the VISA Memory Allocation function.



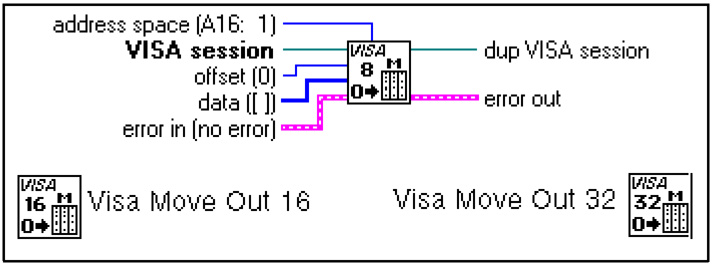
VISA Move In8 / Move In16 / Move In32

Moves a block of data from device memory to local memory in accesses of 8-bits, 16-bits, or 32-bits, respectively.



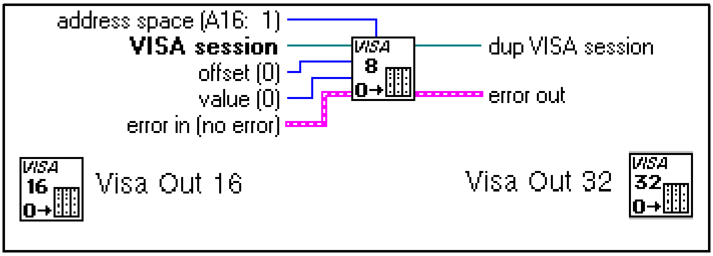
VISA Move Out8 / Move Out16 / Move Out32

Moves a block of data from local memory to device memory in accesses of 8-bits, 16-bits, or 32-bits, respectively.



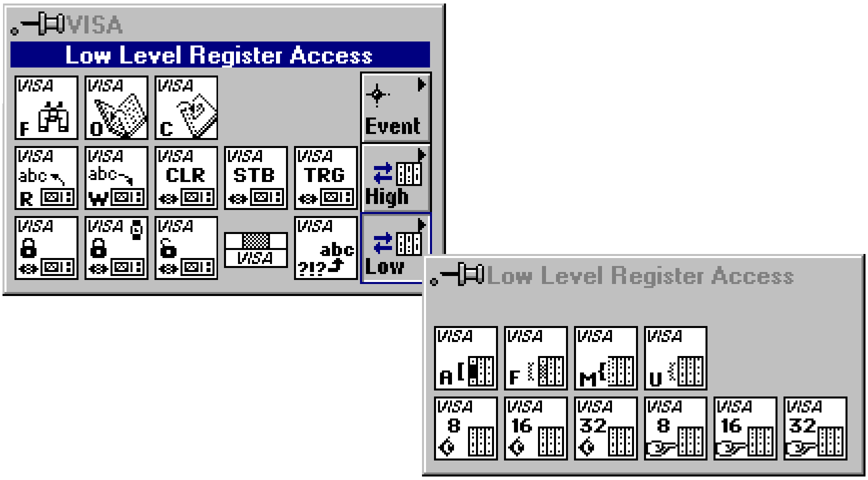
VISA Out8 / Out16 / Out32

Writes 8-bits, 16-bits, or 32-bits of data, respectively, to the specified memory space (assigned memory base + offset).



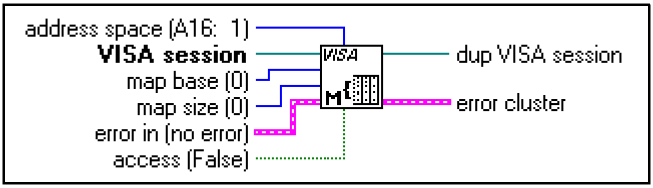
Low Level Register Access Functions

The following section 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.To access the VISA Low Level Register Access functions, pop up on the Low Level icon on the **VISA** palette:



VISA Map Address

Maps in a specified memory space.



VISA Memory Allocation

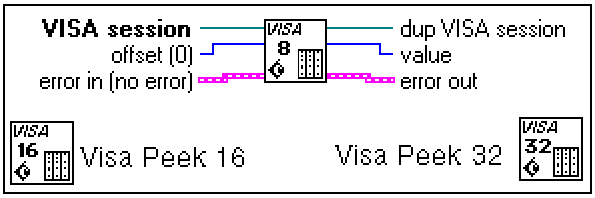
For information on the VISA Memory Allocation function, see the *High Level Register Access Functions* section of this chapter.

VISA Memory Free

For information on the VISA Memory Free function, see the *High Level Register Access Functions* section of this chapter.

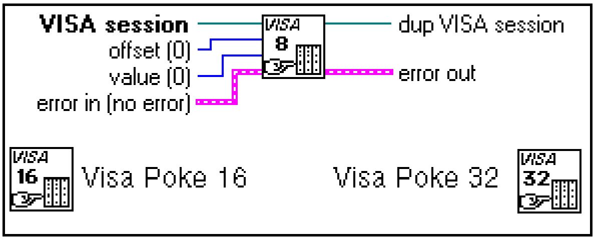
VISA Peek8 / Peek16 / Peek32

Reads an 8-bit, 16-bit, or 32-bit value, respectively, from the specified address.



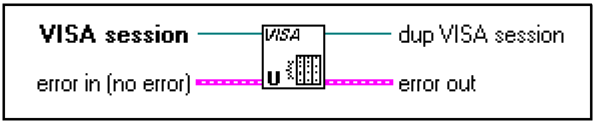
VISA Poke8 / Poke16 / Poke32

Writes an 8-bit, 16-bit, or 32-bit value, respectively, to the specified address.



VISA Unmap Address

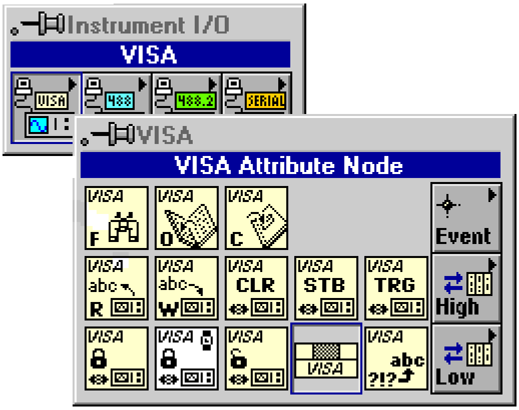
Unmaps memory space previously mapped by VISA Map Address.



VISA Attribute Node

This section describes the VISA Library attributes. The VISA Attribute Node gets and/or sets the indicated attributes. The node is growable; 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).

VISA Attribute Node Descriptions

Each attribute description includes the attribute's 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

Specifies which FDC mode to use (either normal or stream mode).

Fast Data Channel Number

Determines which fast data channel (FDC) will be used to transfer the buffer.

Fast Data Channel Pairs

Specifies use of a channel pair for transferring data; (otherwise, only one channel will be used).

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.

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. 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.

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.

InterfaceType

Specifies the interface type of the given session.

Mainframe Logical Address

Specifies the lowest logical address in the mainframe. If the logical address is not known, `UNKNOWN LA` is returned.

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. `This attribute` is Read/Write until the first time Enable Event is called on a session. Thereafter, this attribute is Read Only.

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 Identification

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.

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.

Slot

Specifies the physical slot location of the VXIbus device. If the slot number is not known, UNKNOWN SLOT is returned.

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.

Termination Character

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.

Termination Character Enable

Determines whether the read operation should terminate when a termination character is received.

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.

Trigger Identifier

Identifier for the current triggering mechanism.

☞ **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.

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.

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.

VXI Commander Logical Address

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. The three types are A16 only, A16/A24, or A16/A32 memory address space.

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.

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

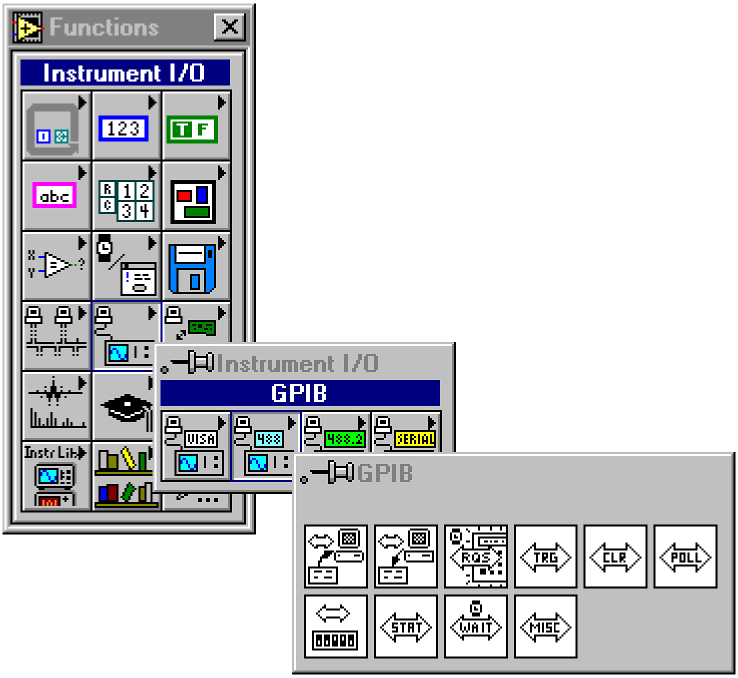
Specifies the size of the region mapped to this window.

Chapter 35

Traditional GPIB Functions

This chapter describes the Traditional GPIB functions.

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



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

Traditional GPIB Function Parameters

Most of the Traditional GPIB functions use the following parameters:

- **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**.

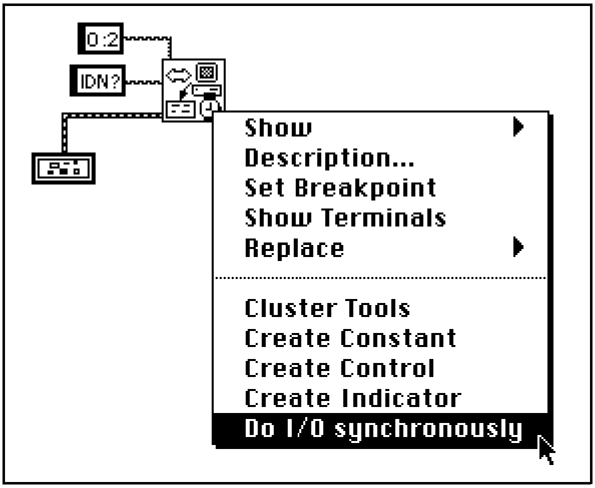
When there are multiple GPIB Controllers that LabVIEW can use, a prefix to the **address string** in the form `ID:address` (or `ID:` if no address is necessary) determines the Controller that a specific function uses. If a Controller ID is not present, the functions assume Controller (or bus) 0.

- **status** 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 GPIB Status in the *GPIB Function Descriptions* section of this chapter for status bit error codes.
- **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. Table 6-3 lists the GPIB error codes. 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 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.

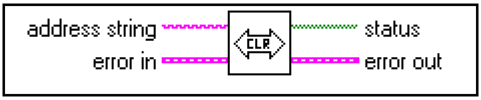


Traditional GPIB Function Descriptions

The following Traditional GPIB functions are available.

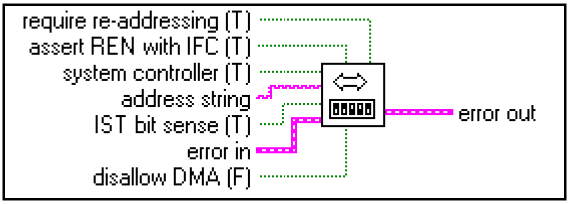
GPIB Clear

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



GPIB Initialization

Configures the GPIB interface at **address string**.



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.

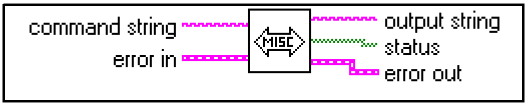


Table 35-1. Command String Functions

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).
GPIB Controller Functions	**Description**
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.

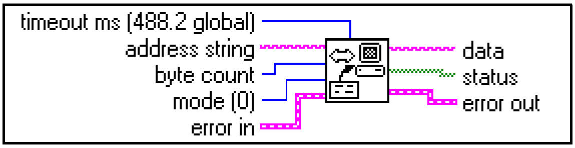
Table 35-1. Command String Functions (Continued)

Device Functions	Description
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.

GPIB Read

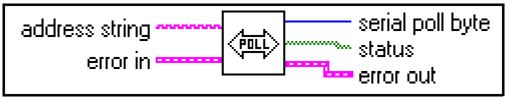
Reads **byte count** number of bytes from the GPIB device at **address string**.



You use the SetTimeOut function to change the default value (the 488.2 global timeout) of **timeout ms**. Initially, **timeout ms** defaults to 10,000. See the description of the SetTimeOut function in Chapter 36, *GPIB 488.2 Functions*, for more information.

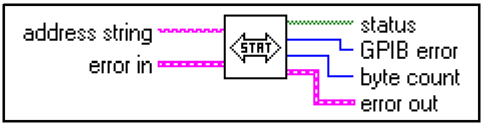
GPIB Serial Poll

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



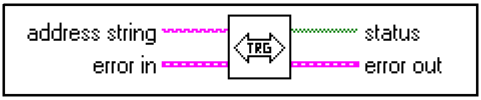
GPIB Status

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



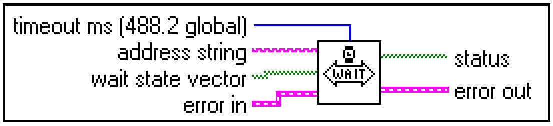
GPIB Trigger

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



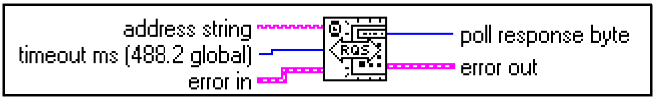
GPIB Wait

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



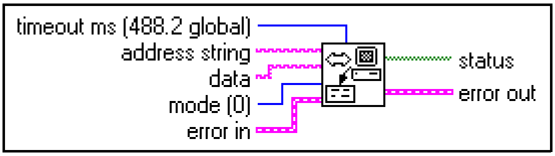
Wait for GPIB RQS

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



GPIB Write

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



GPIB Device and Controller Functions

This section describes the functions listed in the GPIB Misc function description. The device functions send configuration information to a specific instrument (device). The GPIB 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

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

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 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.

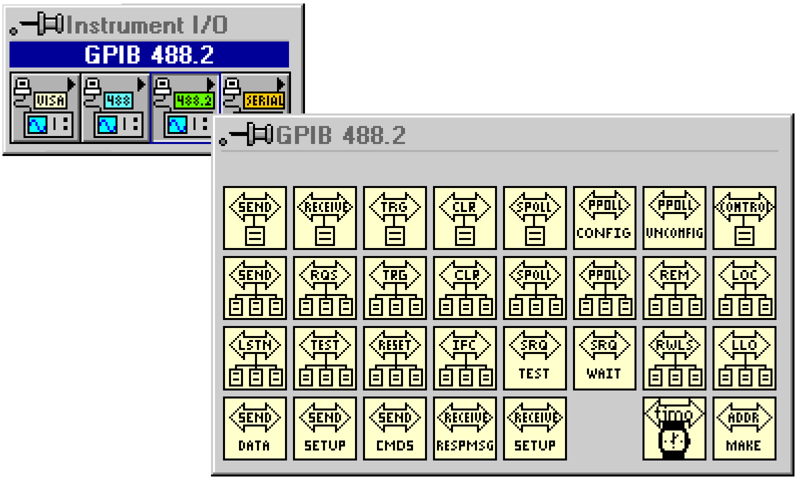
The ESAC error occurs if the Controller is not System Controller.

Chapter 36

GPIB 488.2 Functions

This chapter describes the IEEE 488.2 (GPIB) Functions.

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



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

GPIB 488.2 Common Function Parameters

Most of the GPIB 488.2 Functions use the following parameters:

- **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.

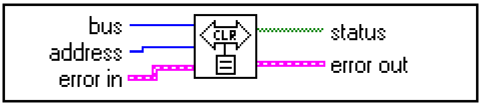
- **bus** refers to the GPIB bus number. If you have only one GPIB interface in your computer, the default bus number is 0. For additional GPIB interfaces, see the software installation instructions included with your GPIB board.
- **byte count** refers to the number of bytes that pass over the GPIB.
- **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 in Chapter 35, *Traditional GPIB Functions*.
- **error in**; **error out**. See the *GPIB Traditional Function Parameters* section of Chapter 35, *Traditional GPIB Functions*.

GPIB 488.2 Function Descriptions (Single-Device Functions)

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.

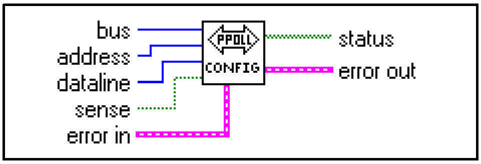
DevClear

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



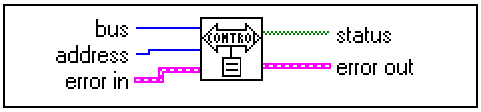
PPollConfig

Configures a device for parallel polls.



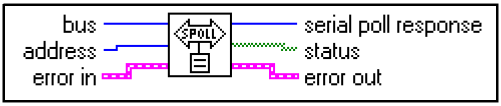
PassControl

Passes control to another device with Controller capability.



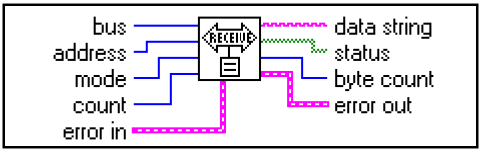
ReadStatus

Serial polls a single device to get its status byte.



Receive

Reads data bytes from a GPIB device.

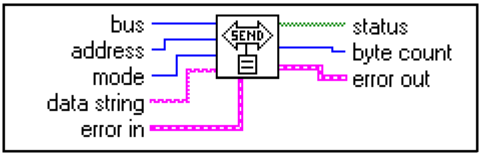


Receive terminates when the function does one of the following:

- 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)

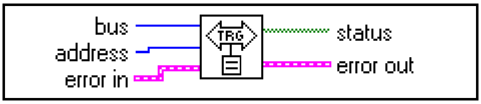
Send

Sends data bytes to a single GPIB device.



Trigger

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



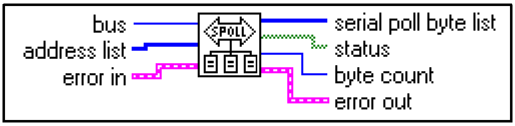
GPIB 488.2 Multiple-Device Function Descriptions

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.

AllSPoll

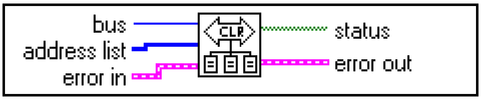
Serial polls all devices.

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.



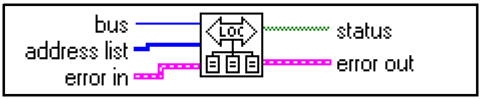
DevClearList

Clears multiple devices simultaneously.



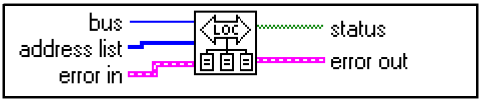
EnableLocal

Enables local mode for multiple devices.



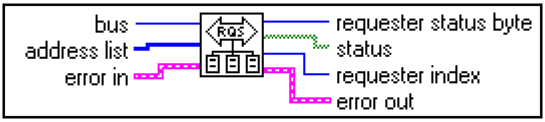
EnableRemote

Enables remote programming of multiple GPIB devices.



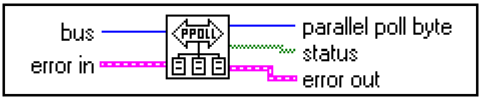
FindRQS

Determines which device is requesting service.



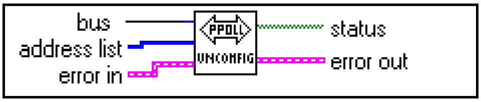
PPoll

Performs a parallel poll.



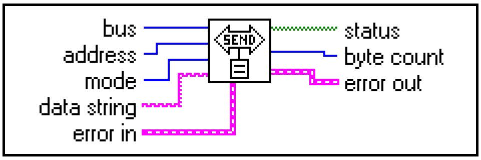
PPollUnconfig

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.



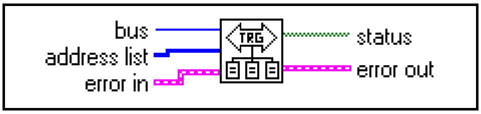
SendList

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.



TriggerList

Triggers multiple devices simultaneously.

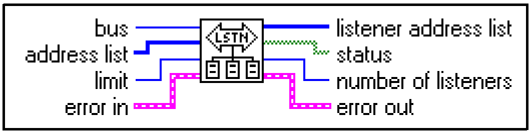


GPIB 488.2 Bus Management Function Descriptions

The bus management functions perform system-wide functions or report system-wide status.

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.

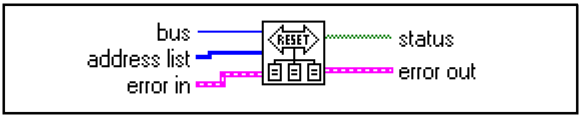


ResetSys

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.

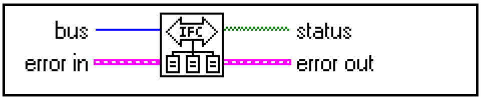
Third, the function sends the *RST message to all devices whose addresses are contained in the **address list** array. This message initializes device-specific functions within each device.



SendIFC

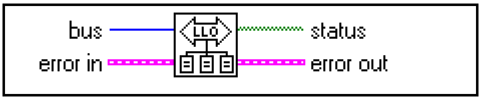
Clears the GPIB functions with Interface Clear (IFC). When you issue the GPIB Device IFC message, the interface functions of all connected devices return to their cleared states.

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.



SendLLO

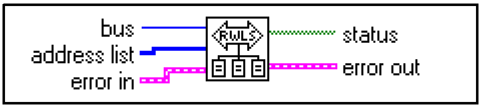
Sends the Local Lockout (LLO) message to all devices. When the function sends the GPIB Local Lockout message, a device cannot independently choose the local or remote state. While Local Lockout is in effect, only the Controller can alter the local or remote state of the devices by sending the appropriate 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.



SetRWLS

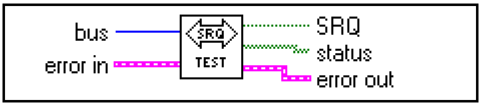
Places particular devices in the Remote With Lockout State. The function sends Remote Enable (REN) to the GPIB devices listed in **address list**. It also places all devices in

Lockout State, which prevents them from independently returning to local programming mode without intervention by the Controller.



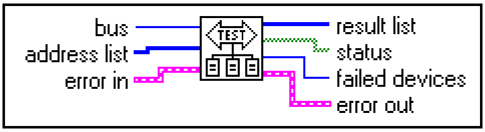
TestSRQ

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.



TestSys

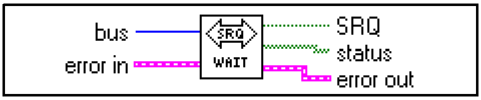
Directs multiple devices to conduct IEEE 488.2 self-tests.



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.

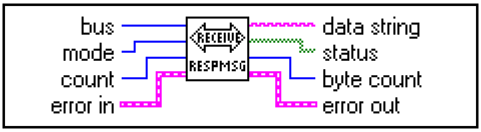


GPIB 488.2 Low-Level I/O Function Descriptions

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.

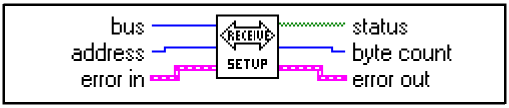
RcvRespMsg

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.



ReceiveSetup

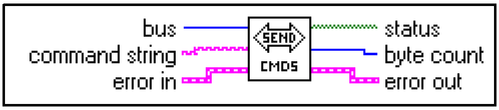
Prepares a device to send data bytes and prepares the GPIB board to read data bytes. After you call this function, you can use a function such as RcvRespMsg to transfer the data from the Talker. In this way, you eliminate the need to re-address the devices between blocks of reads.



SendCmds

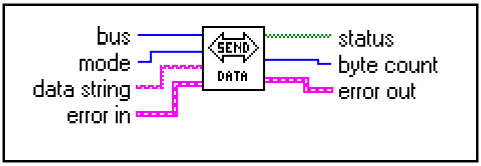
Sends GPIB command bytes.

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.



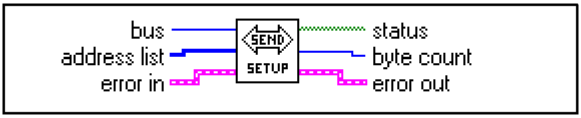
SendDataBytes

Sends data bytes to previously addressed devices.



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.

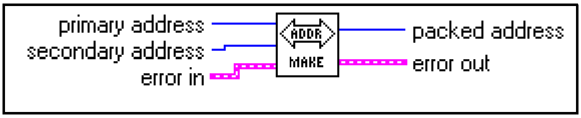


GPIB 488.2 General Function Descriptions

The general functions are useful for special situations.

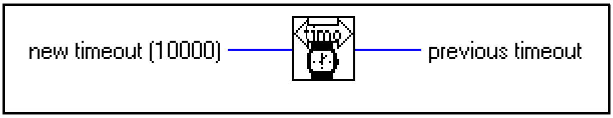
MakeAddr

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



SetTimeOut

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

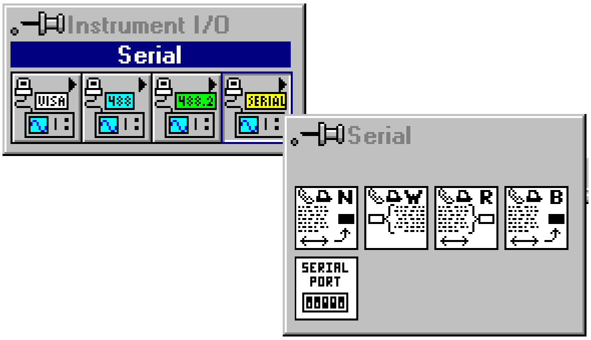


Chapter 37

Serial Port VIs

This chapter describes the VIs for serial port operations.

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



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

Common Serial Port VI Parameters

The following are common Serial Port VI parameters.

Port Number

When you use the serial port VIs under Windows 95 and Windows 3.x, the **port number** parameter can have 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 a Sun SPARCstation under Solaris 1 and on Concurrent PowerMAX, 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/tty00`, 1 to `/dev/tty01`, 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.

Because other vendor's serial port boards can have arbitrary device names, LabVIEW has developed an easy interface to keep the numbering of ports simple. In LabVIEW for Sun, HP-UX, and Concurrent PowerMAX, a configuration option exists to tell LabVIEW how to address the serial ports. LabVIEW supports any board that uses standard UNIX devices. Some manufacturers suggest using `cua` rather than `tty` device nodes with their boards. LabVIEW can address both types of nodes.

The file `.labviewrc` contains the LabVIEW configuration options. To set the devices the serial port VIs use, set the configuration option `labview.serialDevices` to the list of devices you intend to use. For example, the default is:

```
labview.serialDevices:/dev/ttya:/dev/ttyb:/dev/ttyc:...
:/dev/ttyz.
```

☞ **Note:** ***This requires that any third party serial board installation include a method of creating a standard*** `/dev` ***file (node) and that the user knows the name of that file.***

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 the error handler VIs, refer to the *Error Handling Overview* section in Chapter 10, *Time, Dialog and Error Functions*.

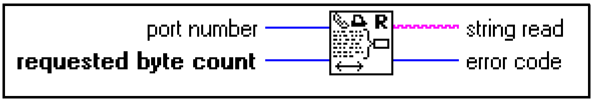
Some error codes returned by the serial port VIs are platform-specific. Please refer to your system documentation for a list of error codes.

Serial Port VI Descriptions

The following Serial Port VIs are available.

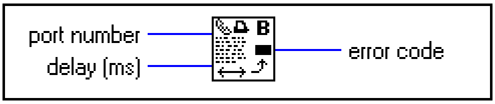
Bytes at Serial Port

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



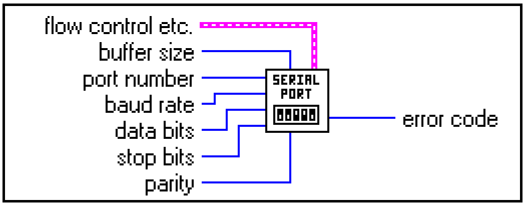
Serial Port Break

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.



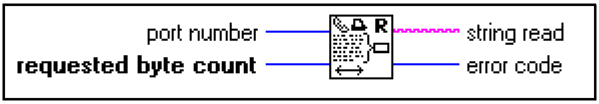
Serial Port Init

Initializes the selected serial port to the specified settings.



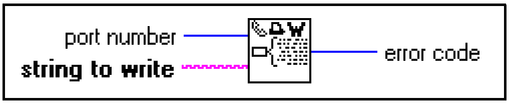
Serial Port Read

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



Serial Port Write

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

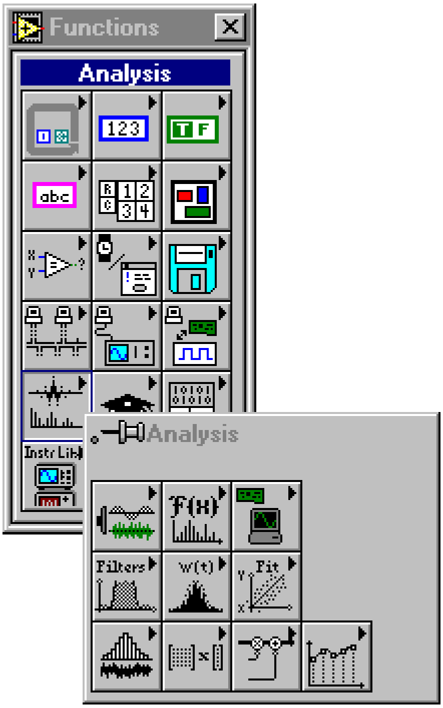


Introduction to Analysis in LabVIEW

Chapter 38

This chapter contains an overview of data analysis and of the LabVIEW analysis VIs, a description of how the VIs are organized, instructions for accessing the VIs and obtaining online help, and a description of analysis VI error reporting.

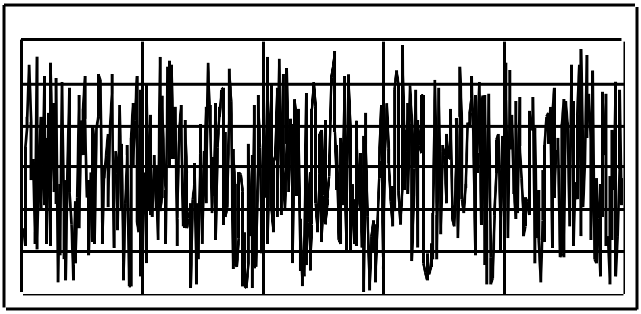
To access the analysis VIs from the block diagram window, choose **Functions»Analysis** and proceed through the hierarchical menus, and select the VI you want. You can place the icon corresponding to that VI in the block diagram and then wire it.



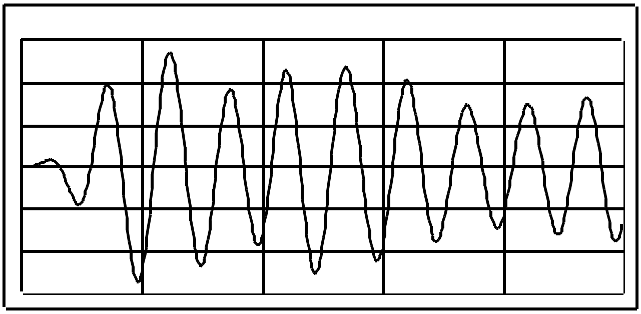
The Importance of Data Analysis

Modern, high-speed floating-point numerical and digital signal processors have become increasingly important to real-time and analysis systems. A few of the many possible applications include biomedical data processing, speech synthesis and recognition, and digital audio and image processing.

The importance of integrating analysis libraries into engineering stations is that the *raw* data, as shown in the following figure, does not always immediately convey useful information. Often you must transform the signal, remove noise disturbances, correct for data corrupted by faulty equipment, or compensate for environmental effects, such as temperature and humidity.



By analyzing and processing the digital data, you can extract the useful information from the noise and present it in a form more comprehensible than the raw data. The following figure shows the processed data.



The LabVIEW block diagram programming approach and the extensive set of LabVIEW analysis VIs simplify the development of analysis applications.

The LabVIEW analysis VIs give you the most recent data analysis techniques using VIs that you can wire together, as shown in the preceding figure, to analyze data. Instead of worrying about implementation details for analysis routines, as you do in conventional programming languages, you can concentrate on solving your data analysis problems.

Full Development System

The base analysis VI library is a subset of the advanced analysis VI library. The base analysis library includes VIs for statistical analysis, linear algebra, and numerical analysis. The advanced analysis library includes more VIs in these areas as well as VIs for signal generation, time and frequency-domain algorithms, windowing routines, digital filters, evaluations, and regressions.

If the VIs in the base analysis library do not satisfy your needs, then you can add the LabVIEW Advanced Analysis Libraries to the G Base Package. Once you upgrade, you will have all the analysis tools available in the Full Development System.

Refer to the chapters that introduce each section for information on how to access a particular Function or VI palette.

Analysis VI Overview

The LabVIEW analysis VIs efficiently process blocks of information represented in digital form. They cover the following major processing areas:

- Pattern generation
- Digital signal processing
- Measurement-based analysis
- Digital filtering
- Smoothing windows
- Probability and Statistical analysis
- Curve fitting

- Linear algebra
- Numerical analysis

The analysis VIs perform numerical operations using the central processing unit (CPU) and a floating-point coprocessor (FPU). Many of the VIs take advantage of the concurrent processing capabilities of the CPU and the FPU, thereby minimizing execution time of data analysis tasks.

The analysis VIs use in-place data processing algorithms. That is, the algorithms allocate minimal data space and process the data within that space. In-place processing minimizes memory requirements, so you can process larger data blocks. The only memory limitation for these VIs is the amount of RAM available in your computer. Refer to your *LabVIEW User Manual* for instructions on configuring the memory allocation for LabVIEW.

The analysis VIs are powerful enough for experts to build sophisticated analysis applications quickly and efficiently. At the same time, they are simple enough for novices to analyze data without being expert programmers in DSP, digital filters, statistics, or numerical analysis.

Analysis VI Organization

After installation, the ten analysis VI libraries appear in the **Functions** palette. These libraries cover the following major processing areas:

- **Signal Generation** contains VIs that generate digital patterns and waveforms.
- **Digital Signal Processing** contains VIs that perform frequency domain transformations, frequency domain analysis, time domain analysis, and other transforms such as the Hartley and Hilbert transforms.
- **Measurement** contains VIs that perform measurement-oriented functions such as single-sided spectrums, scaled windowing, and peak power and frequency estimation.
- **Filters** contains VIs that perform IIR, FIR, and nonlinear, digital filtering functions.
- **Windows** contains VIs that perform data windowing.
- **Probability and Statistics** contains VIs that perform descriptive statistics functions, such as identifying the mean or the standard

deviation of a set of data, as well as inferential statistics functions for probability and analysis of variance (ANOVA).

- **Curve Fitting** contains VIs that perform curve fitting functions and interpolations.
- **Linear Algebra** contains VIs that perform algebraic functions for real and complex vectors and matrices.
- **Array Operations** contains VIs that perform common, one- and two-dimensional numerical array operations, such as linear evaluation and scaling.
- **Additional Numerical Methods** contains VIs that use numerical methods to perform root-finding, numerical integration, and peak detection.

You can reorganize the folders and the VIs to suit your needs and applications. You can also rebuild the original structure by removing the VIs from your hard disk and then reinstalling them from the distribution disks.

Notation and Naming Conventions

To help you identify the type of parameters and operations, this section of the manual uses the following notation and naming conventions unless otherwise specified in a VI description. Although there are a few scalar functions and operations, most of the analysis VIs process large blocks of data in the form of one-dimensional arrays (or vectors) and two-dimensional arrays (or matrices).

Normal lower case letters represent scalars or constants. For example,

a,

π,

$b = 1.234$.

Capital letters represent arrays. For example,

X,

A,

$Y = a X + b$.

In general, X and Y denote 1D arrays, and A, B, and C represent matrices.

Array indexes in LabVIEW are zero-based. The index of the first element in the array, regardless of its dimension, is zero. The following sequence of numbers represents a 1D array *X* containing *n* elements.

$$X = \{x_0, x_1, x_2, ..., x_{n-1}\}$$

The following scalar quantity represents the i^{th} element of the sequence *X*.

$$x_i, \quad 0 \le i < n$$

The first element in the sequence is x_0 and the last element in the sequence is $x_{n\text{-}1}$, for a total of *n* elements.

The following sequence of numbers represents a 2D array containing *n* rows and *m* columns.

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \dots & a_{0m-1} \\ a_{10} & a_{11} & a_{12} & \dots & a_{1m-1} \\ a_{20} & a_{21} & a_{22} & \dots & a_{2m-1} \\ : & : & : & : & : \\ a_{n-10} & a_{n-11} & a_{n-2} & \dots & a_{n-1m-1} \end{bmatrix}$$

The total number of elements in the 2D array is the product of *n* and *m*. The first index corresponds to the row number, and the second index corresponds to the column number. The following scalar quantity represents the element located on the i^{th} row and the j^{th} column.

$$a_{i\,j}, 0 \le i < n \text{ and } 0 \le j < m$$

The first element in *A* is $a_{0\,0}$ and the last element is $a_{n\text{-}1\;m\text{-}1}$.

Unless otherwise specified, this manual uses the following simplified array operation notations.

Setting the elements of an array to a scalar constant is represented by

$X = a$,

which corresponds to the sequence

$X = \{a, a, a, \dots, a\}$

and is used instead of

$x_i = a$, for $i = 0, 1, 2, \ldots, n\text{-}1$.

Multiplying the elements of an array by a scalar constant is represented by

$Y = a\,X$,

which corresponds to the sequence

$Y = \{a\,x_0, a\,x_1, a\,x_2, \ldots, a\,x_{n\text{-}1}\}$

and is used instead of

$y_i = a\,x_i$, for $i = 0, 1, 2, \ldots, n\text{-}1$.

Similarly, multiplying a 2D array by a scalar constant is represented by

$B = k\,A$,

which corresponds to the sequence

$$B = \begin{bmatrix} ka_{00} & ka_{01} & ka_{02} & \ldots & ka_{0m-1} \\ ka_{10} & ka_{11} & ka_{12} & \ldots & ka_{1m-1} \\ ka_{20} & ka_{21} & ka_{22} & \ldots & ka_{2m-1} \\ : & : & : & : & : \\ ka_{n-10} & ka_{n-11} & ka_{n-12} & \ldots & ka_{n-1m-1} \end{bmatrix}$$

and is used instead of

$b_{ij} = k\,a_{ij}$, for $i = 0, 1, 2, \ldots, n\text{-}1$ and $j = 0, 1, 2, \ldots, m\text{-}1$.

Empty arrays are possible in LabVIEW. An array with no elements is an empty array and is represented by

Empty = NULL = Ø = { } .

In general, operations on empty arrays result in empty, output arrays or undefined results.

Sampling Signals

To use digital signal processing techniques, you must convert an analog signal into its digital representation. This section includes only a brief discussion of the notation that represents a digital signal. This section does not discuss the mathematical background or problems associated with sampling techniques.

Consider an analog signal $x(t)$ and the sampling interval Δt. The signal $x(t)$ can be represented by the discrete sequence of samples

$\{x(0), x(\Delta t), x(2\Delta t), x(3\Delta t), \ldots, x(k\Delta t), \ldots \}$.

Because Δt establishes only the sampling rate and has no bearing on the actual sampled (digitized) value, the sample at

$t = i\Delta t$, for $i = 0, 1, 2, \ldots$

corresponds to the i^{th} element in the sequence.

Thus,

$x_i = x(i\Delta t)$

and $x(t)$ can be represented by the sequence X whose values are

$X = \{x_0, x_1, x_2, x_3, \ldots, x_{\mathrm{k}}, \ldots \}$.

If n samples are obtained from the signal $x(t)$, then the sequence

$X = \{x_0, x_1, x_2, x_3, \ldots, x_{n\text{-}1} \}$

is the digital representation or the sampled version of $x(t)$.

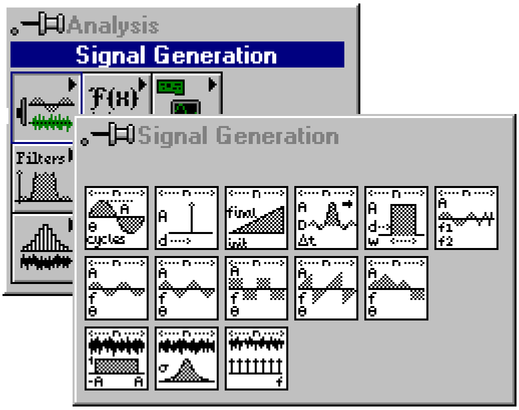
Chapter 39

Analysis Signal Generation VIs

This chapter describes the VIs that generate one-dimensional arrays with specific waveform patterns.

You can combine these VIs with the arithmetic functions discussed in Chapter 4, *Numeric Functions,* to generate more elaborate waveforms. For example, if you want to generate an amplitude modulated pulse, you multiply a pulse pattern by a sinusoidal pattern.

To access the Signal Analysis palette, select **Function»Analysis»Signal Generation**. The following illustration shows the options that are available on the Signal Analysis palette.



For examples of how to use the signal generation VIs, see the examples located in `examples\analysis\sigxmpl.llb`.

Normalized Frequency

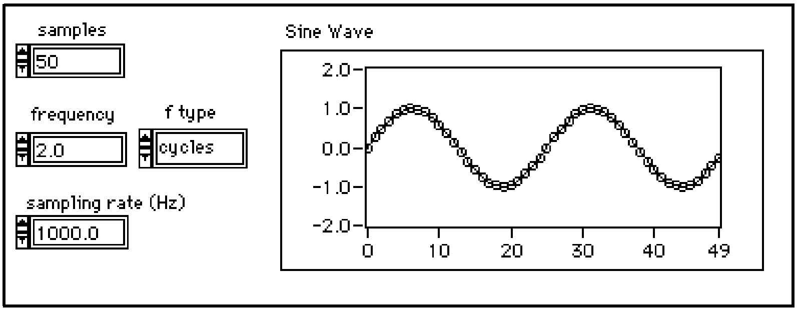
Some of the Signal Generation VIs use an input frequency control (*f*) that is assumed to use normalized frequency units of cycles per sample. Its reciprocal (1/*f*) gives you the number of times that the signal is sampled in one cycle. This frequency ranges from 0 to 1.0, which corresponds to a real frequency range of 0 to the sampling rate. This frequency also wraps around 1.0, so that a normalized frequency of 1.1 is equivalent to 0.1.

For example, if a signal is sampled at the *Nyquist* rate (fs/2), it is sampled twice per cycle. This corresponds to a normalized frequency of 1/2 samples/cycle that is less than or equal to 0.5 cycles/sample.

If you use some of these VIs, you must convert your frequency units to the normalized units of cycles/sample. You must use these normalized units with the following VIs:

- Sine Wave
- Square Wave
- Sawtooth Wave
- Triangle Wave
- Arbitrary Wave
- Chirp Pattern

If you are used to working in frequency units of cycles, you can convert cycles to cycles/sample by dividing cycles by the number of samples generated. The following illustration shows an example of the Sine Wave VI generating two cycles of a sine wave.



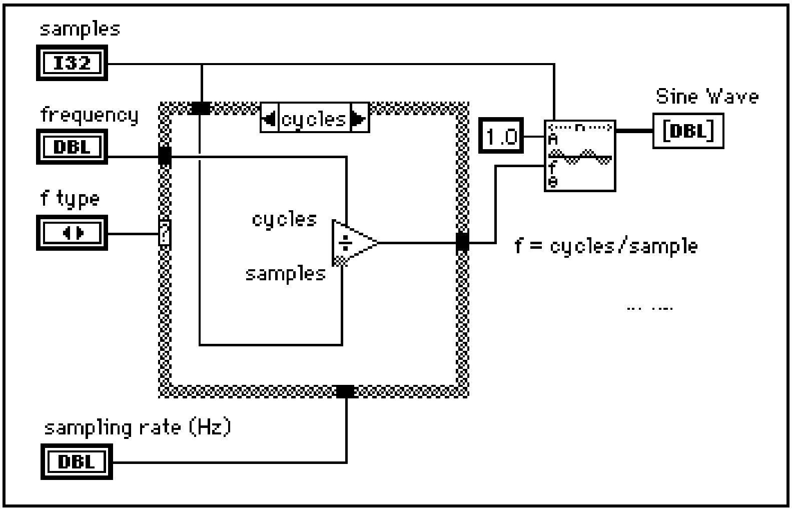
samples: 50

frequency: 2.0

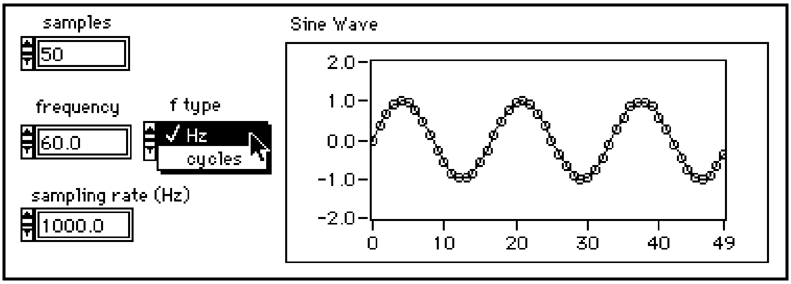
f type: cycles

sampling rate (Hz): 1000.0

The following illustration shows the block diagram for converting cycles to cycles/sample.



However, you may need to use frequency units of Hz (cycles/s). If you need to convert to Hz (or cycles/s) to cycles/sample, divide your frequency in cycles/s by the sampling rate given in samples/s. The following illustration shows an example of the Sine Wave VI generating a 60 Hz sine signal.


samples 50

frequency 60.0

f type ✓ Hz

sampling rate (Hz) 1000.0

Figure 39-1. Front Panel Example

The following illustration shows the block diagram for generating a sine signal.

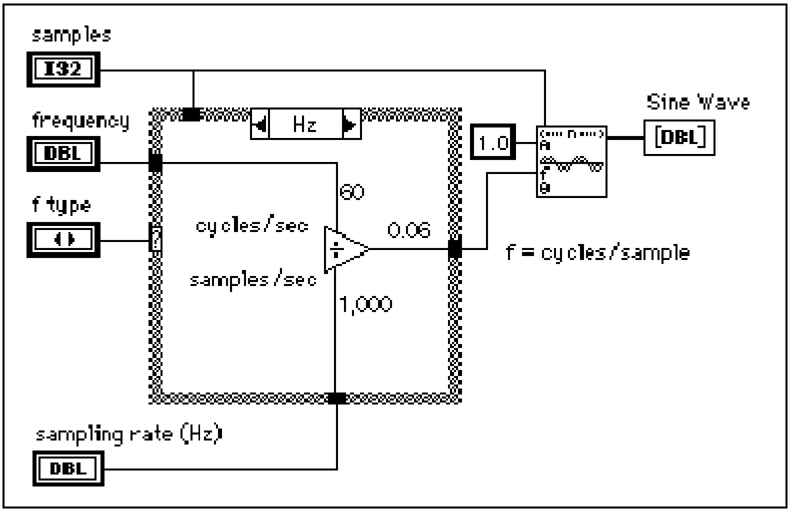


Figure 39-2. Block Diagram example

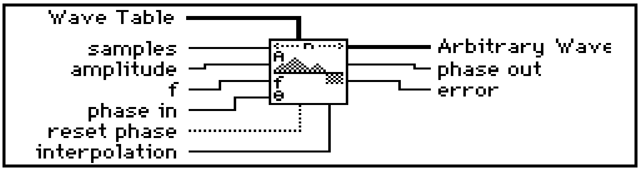
For example, build a VI with the Front Panel and Block Diagram, as illustrated in the Figures 39-1 and Figure 39-2 above. Select a frequency of 2 cycles and the number of samples of 100. 2 cycles appear on the plot. Change the number of samples to 150, 200, and 250 and 2 cycles remain. If you keep the number of samples equal to 100 and the number of cycles to 3, 4, and 5, there are 3, 4, and 5 cycles, respectively. Therefore, when you choose the frequency in number of cycles, you will see that many cycles within the plot.

Signal Generation VI Descriptions

The following Signal Generation VIs are available.

Arbitrary Wave

Generates an array containing an arbitrary wave.



If the sequence y represents **Arbitrary Wave**, then the VI generates the pattern according to the following formula:

$$y[i] = a * \mathrm{arb}(\mathrm{phase}[i]), \quad \text{for } i = 0, 1, 2, ..., n\text{-}1,$$

where a is the **amplitude**, n is the number of **samples**,

$$\mathrm{arb}(\mathrm{phase}[i]) = \mathrm{WT}(\mathrm{phase}[i] \text{ modulo } 360)*m/360)$$

where m is the size of the **Wave Table** array

If **interpolation** = 0 (no interpolation), then $WT(x)$ = **Wave Table**[int(x)].

If **interpolation** = 1 (linear interpolation), then $WT(x)$ is equal to the linearly interpolated value of **Wave Table**[int(x)] and **Wave Table**[(int(x)+1) modulo m].

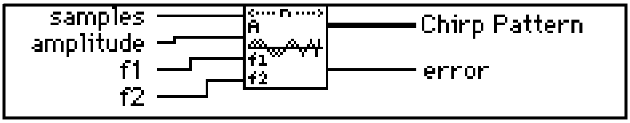
phase[i] = initial_phase + **f***360.0*i, where **f** is the frequency in normalized units of cycles/sample, initial_phase is **phase in** if **reset phase** is true, or initial_phase is the **phase out** from the previous execution of this VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from an arbitrary wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of this VI produce the output **Arbitrary Wave** array containing the next **samples** of the arbitrary wave.

phase out is set to phase[n], and this reentrant VI uses this value as its new **phase in** if **reset phase** is false the next time the VI executes.

Chirp Pattern

Generates an array containing a chirp pattern.



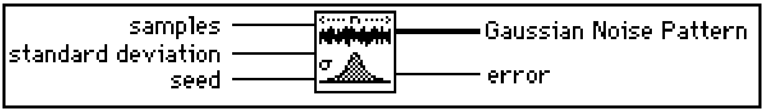
If the sequence Y represents **Chirp Pattern**, the VI generates the pattern according to the following formula:

$$y_i = A * \sin((a/2 \text{ i} + b)\ i), \quad \text{for } i = 0, 1, 2, ..., n\text{-}1,$$

where A is the **amplitude**, $a = 2\pi(\mathbf{f2}\text{-}\mathbf{f1})/n$, $b = 2\pi\mathbf{f1}$, **f1** is the beginning frequency in normalized units of cycles/sample, **f2** is the ending frequency in normalized units of cycles/sample, and n is the number of **samples**.

Gaussian White Noise

Generates a Gaussian-distributed, pseudorandom pattern whose statistical profile is (μ, σ) = (0, *s*), where *s* is the absolute value of the specified **standard deviation**.



To generate the pattern, the VI uses a modified version of the Very-Long-Cycle random number generator algorithm based upon the Central Limit Theorem. Given that the probability density function, *f*(*x*), of the Gaussian-distributed **Gaussian Noise Pattern** is:

$$f(x) = \frac{1}{\sqrt{2\pi}s} e^{\left(-\frac{1}{2}\right)\left(\frac{x}{s}\right)^2} ,$$

where *s* is the absolute value of the specified **standard deviation** and that you can compute the expected values, *E*{•}, using the formula:

$$E(x) = \int_{-\infty}^{\infty} x(f(x))dx ,$$

then the expected mean value, μ, and the expected standard deviation value, σ, of the pseudorandom sequence are:

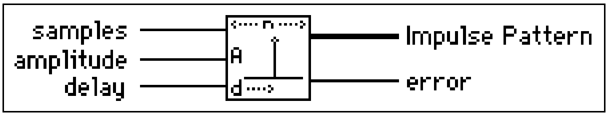
$$\mu = E\{x\} = 0,$$

$$\sigma = [E\{(x-\mu)^2\}]^{1/2} = s .$$

The pseudorandom sequence produces approximately 2^{90} samples before the pattern repeats itself.

Impulse Pattern

Generates an array containing an impulse pattern.



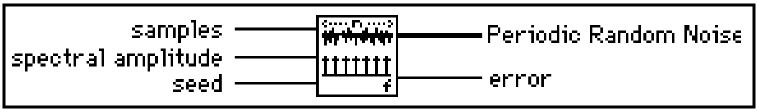
If the **Impulse Pattern** is represented by the sequence X, the VI generates the pattern according to the following formula:

$$x_i = \begin{cases} a \ \text{ if } i = d & \text{for i = 0, 1, 2, . . ., n-1} \\ 0 \ \ \text{elsewhere} & \end{cases}$$

where *a* is the **amplitude**, *d* is the **delay**, and *n* is the number of **samples**.

Periodic Random Noise

Generates an array containing periodic random noise (PRN).



The output array contains all frequencies which can be represented with an integral number of cycles in the requested number of **samples**. Each frequency-domain component has a magnitude of **spectral amplitude** and random phase.

You can think of the output array of PRN as a summation of sinusoidal signals with the same amplitudes but with random phases. The unit of **spectral amplitude** is the same as the output **Periodic Random Noise**, and is a linear measure of amplitude, similar to other signal generation VIs.

The VI generates the same periodic random sequence for a given positive **seed** value. The VI does not reseed the random phase generator if **seed** is negative.

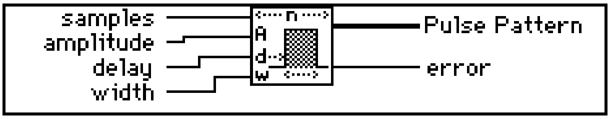
The output sequence is bounded by an amplitude of **spectral amplitude** $* \frac{\textbf{samples}}{2}$.

You can use PRN to compute the frequency response of a linear system in one time record instead of averaging the frequency response over several time records, as you must for nonperiodic random noise sources.

You do not need to window PRN before performing spectral analysis; PRN is self-windowing and, therefore, has no spectral leakage because PRN contains only integral-cycle sinusoids.

Pulse Pattern

Generates an array containing a pulse pattern.



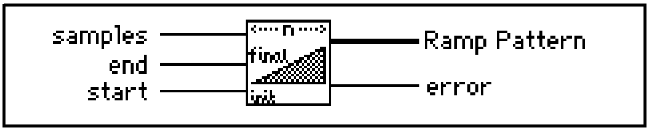
If the sequence *X* represents **Pulse Pattern**, the VI generates the pattern according to the following formula:

$$x_i = \begin{cases} a & \text{if } d \le i < (d + w) \quad \text{for i = 0, 1, 2, . . ., n-1.} \\ 0.0 & \text{elsewhere} \end{cases}$$

where *a* is the **amplitude**, *d* is the **delay**, *w* is the **width**, and *n* is the number of **samples**.

Ramp Pattern

Generates an array containing a ramp pattern.



If the sequence *X* represents **Ramp Pattern**, the VI generates the pattern according to the formula:

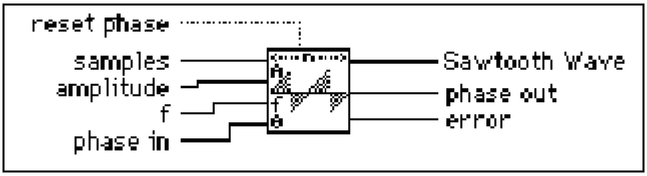
$x_i = x_0 + i\Delta x$ for $i = 0, 1, 2,\ldots, n\text{-}1$,

where $\Delta x = \dfrac{x_{n-1} - x_0}{n-1}$, $x_{n\text{-}1}$ is the **end**, x_0 is the **start**, and *n* is the number of **samples**.

The does not impose conditions on the relationship between **start** and **end**. Therefore, it can generate ramp-up and ramp-down patterns.

Sawtooth Wave

Generates an array containing a sawtooth wave.



If the sequence *Y* represents **Sawtooth Wave**, the VI generates the pattern according to the following formula:

$$y[i] = a * \text{sawtooth}(\text{phase}[i]), \text{ for } i = 0, 1, 2,..., n\text{-}1,$$

where *a* is the **amplitude**, *n* is the number of **samples**,

$$\text{sawtooth (phase}[i]) = \begin{cases} \dfrac{p}{180.0} & 0 \le p < 180 \\ \dfrac{p}{180.0} - 2.0 & 180 \le p < 360 \end{cases}$$

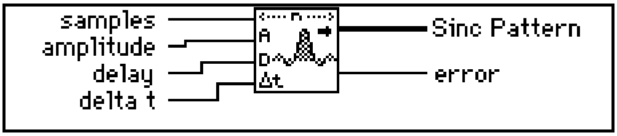
p = phase[*i*] modulo 360.0, phase[*i*] = initial_phase + **f***360.0**i*, **f** is the frequency in normalized units of cycles/sample; initial_phase is **phase in** if **reset phase** is true; or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from a sawtooth wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of the VI produce the output **Sawtooth Wave** array containing the next **samples** of a sawtooth wave.

phase out is set to phase[*n*], and, if **reset phase** is false, the next time the VI executes this reentrant VI uses this value as its new **phase in**.

Sinc Pattern

Generates an array containing a sinc pattern.



If the sequence Y represents **Sinc Pattern**, the VI generates the pattern according to the following formula:

$y_i = a\,\mathrm{sinc}(i\Delta t - d)$, for *i* = 0, 1, 2,…, *n*-1,

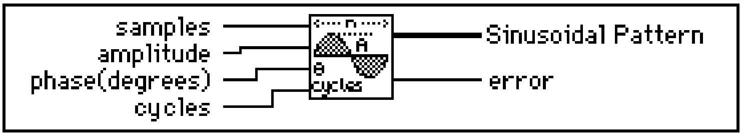
where $\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$, *a* is the **amplitude**, Δ*t* is the sampling interval **delta t**, *d* is the **delay**, and *n* is the number of **samples**.

The main lobe of the sinc function, sinc(*x*), is the part of the sinc curve bounded by the region -1 ≤ *x* ≤ 1.

When |*x*| = 1, the sinc(*x*) = 0.0, and the peak value of the sinc function occurs when *x* = 0. Using l'Hôpital's Rule, you can show that sinc(0) = 1 and is its peak value. Thus, the main lobe is the region of the sinc curve encompassed by the first set of zeros to the left and the right of the sinc value.

Sine Pattern

Generates an array containing a sinusoidal pattern.



If the sequence *Y* represents **Sinusoidal Pattern**, the VI generates the pattern according to the following formula:

$y_i = a\sin(x_i)$ for *i* = 0, 1, 2,…, *n*-1,

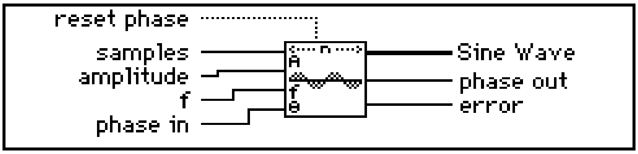
where

$x_i = \frac{2\pi ik}{n} + \frac{\pi\phi_0}{180}$, *a* is the **amplitude**, *k* is the number of **cycles** in the pattern,

ϕ_0 is the initial **phase (degrees)**, and *n* is the number of **samples**.

Sine Wave

Generates an array containing a sine wave.



If the sequence *Y* represents **Sine Wave**, the VI generates the pattern according to the following formula:

$$y_i = a * \sin(\text{phase}[i]), \text{ for } i = 0, 1, 2, ..., n\text{-}1,$$

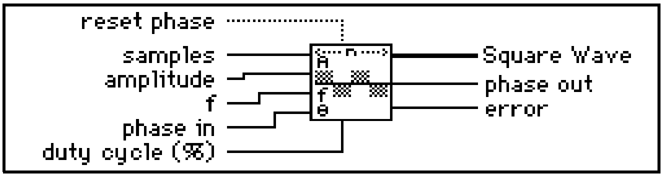
where *a* is the **amplitude** and phase[*i*] = initial_phase + **f***360.0**i*; **f** is the frequency in normalized units of cycles/sample; initial_phase is **phase in** if **reset phase** is true; or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from a sine wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of the VI produce the output **Sine Wave** array containing the next **samples** of a sine wave.

phase out is set to phase[*n*], and if **reset phase** is false the next time the VI executes, this reentrant VI uses this value as the new **phase in**.

Square Wave

Generates an array containing a square wave.



If the sequence *Y* represents **Square Wave**, the VI generates the pattern according to the following formula:

$$y_i = a * \text{square}(\text{phase}[i]), \text{ for } i = 0, 1, 2, ..., n\text{-}1,$$

where *a* is the **amplitude**; *n* is the number of **samples**;

$$\text{square}(\text{phase}[i]) = \begin{cases} 1.0 & 0 \le p < \left(\frac{\text{duty}}{100}360\right) \\ -1.0 & \left(\frac{\text{duty}}{100}360\right) \le p < 360 \end{cases},$$

where *p* = phase[*i*] modulo 360.0, *duty* = **duty cycle**,
phase[*i*] = initial_phase + **f***360.0**i*; **f** is the frequency in normalized units of

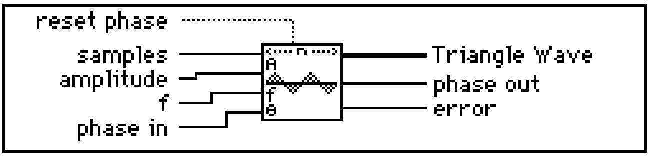
cycles/sample, initial_phase is **phase in** if **reset phase** is true; or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from a square wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of this VI produce the output **Square Wave** array containing the next **samples** of a square wave.

phase out is set to phase[*n*], and if **reset phase** is false the next time the VI executes, this reentrant VI uses this value as its new **phase in**.

Triangle Wave

Generates an array containing a triangle wave.



If the sequence *Y* represents **Triangle Wave**, the VI generates the pattern according to the following formula:

$$y_i = a * \mathrm{tri}(\mathrm{phase}[i]), \text{ for } i = 0, 1, 2, \ldots, n\text{-}1$$

where *a* is the **amplitude**; *n* is the number of **samples**;

$$\mathrm{tri}(\mathrm{phase}[\iota]) = \begin{cases} \dfrac{p}{90} & 0 \le p < 90 \\ 2 - \dfrac{p}{90} & 90 \le p < 270 \\ \dfrac{p}{90} + 4 & 270 \le p < 360 \end{cases}$$

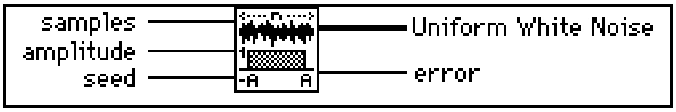
where *p* = (phase[*i*] modulo 360.0); phase[*i*] = initial_phase + **f***360.0**i*; **f** is the frequency in normalized units of cycles/sample; initial_phase is **phase in** if **reset phase** is true; or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from a triangle wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of the VI produce the output **Triangle Wave** array containing the next **samples** of a triangle wave.

phase out is set to phase[*n*], and if **reset phase** is false the next time the VI executes, this reentrant VI uses this value as its new **phase in**.

Uniform White Noise

Generates a uniformly distributed, pseudorandom pattern whose values are in the range [-*a:a*], where *a* is the absolute value of **amplitude**.



The VI generates the pseudorandom sequence using a modified version of the Very-Long-Cycle random number generator algorithm. Given that the probability density function, *f*(*x*), of the uniformly distributed **Uniform White Noise** is

$$f(x) = \begin{cases} \dfrac{1}{2a} & \text{if } -a \le x \le a \\ 0 & \text{elsewhere} \end{cases}$$

where *a* is the absolute value of the specified **amplitude**, and given that you can compute the expected values, $E\{\bullet\}$, using the formula:

$$E(x) = \int_{-\infty}^{\infty} x(f(x))dx$$

, then the expected mean value, μ, and the expected standard deviation value, σ, of the pseudorandom sequence are:

$$\mu = E\{x\} = 0,$$

$$\sigma = [E\{(x-\mu)^2\}]^{1/2} = \frac{a}{\sqrt{3}} \approx 0.57735a \quad .$$

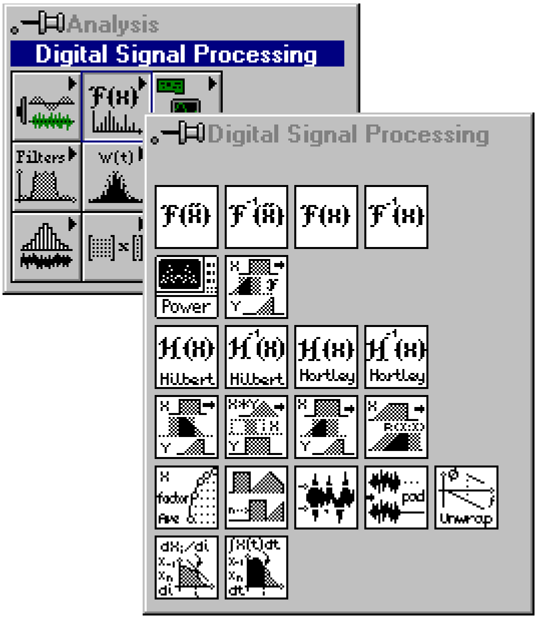
The pseudorandom sequence produces approximately 2^{90} samples before the pattern repeats itself.

Chapter 40

Analysis Digital Signal Processing VIs

This chapter describes the VIs that process and analyze an acquired or simulated signal. The digital signal processing VIs perform frequency domain transformations, frequency domain analysis, time domain analysis, and other transforms, such as the Fourier, Hartley, and Hilbert transforms.

To access the Digital Signal Processing palette, select **Function»Analysis»Digital Signal Processing**. The following illustration shows the options that are available on the Signal Analysis palette.



For examples of how to use the digital signal processing VIs, see the examples located in `examples\analysis\dspxmpl.llb`.

The Fast Fourier Transform (FFT)

The Fourier transform establishes the relationship between a signal and its representation in the frequency domain. The Fourier transform is a powerful analysis tool for spectral analysis, applied mechanics, acoustics, medical imaging, numerical analysis, instrumentation, and telecommunications.

The definition of the Fourier transform of a signal $x(t)$ is

$$X(f) = F(x\{t\}) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt, \tag{40-1}$$

and the inverse Fourier transform of a signal $X(f)$ is

$$x(t) = F^{-1}\{X(f)\} = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft}dt. \tag{40-2}$$

A notation often used to indicate that the signals $x(t)$ and $X(f)$ are a Fourier transform pair and are related via the Fourier transform is:

$$x(t) \Leftrightarrow X(f). \tag{40-3}$$

To derive the discrete representation of the Fourier transform equations, equations (40-1) and (40-2), sample the Fourier transform pair in equation (40-3) using the following sampling relationships:

$$\Delta t = \frac{1}{f_s} \quad \Delta f = \frac{f_s}{n}$$

where Δt is the sampling interval, Δf is the frequency resolution, f_s is the sampling frequency, and n is the number of samples in both the time and frequency domain.

Thus, the discrete transform pair

$$x_i \Leftrightarrow X_k \tag{40-4}$$

is obtained and the discrete Fourier transform is given by

$$X_k = \sum_{i=0}^{n-1} x_i e^{-j2\pi ik/n} \Delta t \qquad (40\text{-}5)$$

and the inverse by

$$x_i = \sum_{i=0}^{n-1} X_k e^{j2\pi ik/n} \Delta f. \qquad (40\text{-}6)$$

X_k in equation (40-5) represents an amplitude spectral density. Multiply the right-hand side of equation (40-5) by the frequency resolution Δf *to* arrive at the amplitude spectrum. This amplitude spectrum is the final form of the DFT and inverse DFT, given by equations (40-7) and (40-8), respectively. Notice that the DFT is independent of the sampling rate.

$$X_k = \sum_{i=0}^{n-1} x_i e^{-j2\pi ik/n} \quad \text{for } k = 0, 1, 2, \ldots, n\text{-}1 \qquad (40\text{-}7)$$

$$x_i = \frac{1}{n} \sum_{k=0}^{n-1} X_k e^{j2\pi ik/n} \quad \text{for } i = 0, 1, 2, \ldots, n\text{-}1. \qquad (40\text{-}8)$$

Direct implementation of the DFT requires approximately n^2 complex operations, and until recently, it was a time-consuming process. However, when the size of the sequence is

$n = 2^m$ for $m = 1, 2, 3, \ldots,$

you can implement the computation of the DFT with approximately $n \log_2(n)$ operations. DSP literature refers to these algorithms as fast Fourier transforms (FFTs).

☞ **Note:** ***The advantages of the FFT include its speed and memory efficiency because the VI performs the transform in place. The size of the input sequence, however, must be a power of 2. The DFT can efficiently process any size sequence, but the DFT is slower than the FFT and uses more memory, because it must store intermediate results during processing.***

Furthermore, with the aid of the FFT, you can find the DFT of any size sequence in approximately $3n\log_2(n)$ operations where n is the next power of 2 that accommodates intermediate results. You can find a more detailed explanation of FFT theory in most introductory texts on DSP.

The algorithm implemented in the LabVIEW analysis VIs is known as the split-radix algorithm. This algorithm has a form similar to the radix-4 algorithms with the efficiency of radix-8 algorithms. The split-radix algorithm requires the least number of multiplications among the radix-2, radix-4, and mixed-radix algorithms.

This manual uses the following notation to denote the discrete Fourier transform of a sequence *x:*

$X = F\{x\}$,

and

$x = F^{-1}\{X\}$

to denote the discrete inverse Fourier transform. The Fourier transform always results in a complex output sequence, and the input sequence can be either *real* or *complex*. Unless otherwise specified, two real sequences represent the complex sequences. If X is a complex sequence, then:

$X_{\mathrm{Re}} = \mathrm{Re}\{X\}$

represents the real part of the complex sequence X,

$X_{\mathrm{Im}} = \mathrm{Im}\{X\}$

represents the imaginary part of the complex sequence X, and

$X = X_{\mathrm{Re}} + j\,X_{\mathrm{Im}} = \mathrm{Re}\{X\} + j\,\mathrm{Im}\{X\}$.

If the i/p signal is real, the FT is symmetric.

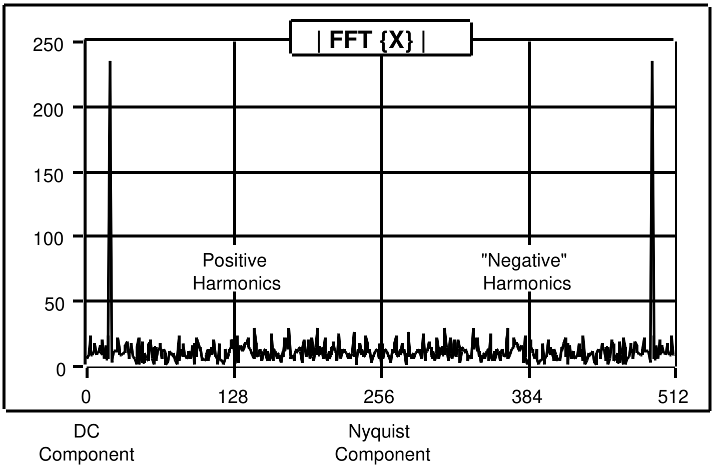
If the i/p signal is complex, the FT is not symmetric.

If n=8, let $k = \frac{n}{2}$. The following table shows the format of the complex output sequence.

Array Element	Interpretation
Y_0	DC component
Y_1	1st harmonic or fundamental
Y_2	2nd harmonic
Y_3	3rd harmonic
. . .	. . .
Y_{k-2}	(k-2)th harmonic
Y_{k-1}	(k-1)th harmonic
Y_k	Nyquist harmonic
$Y_{k+1} = \mathrm{Y}_{n-(k-1)} = Y_{-(k-1)}$	- (k-1)th harmonic*
$Y_{k+2} = Y_{n-(k-2)} = Y_{-(k-2)}$	- (k-2)th harmonic*
. . .	. . .
Y_{n-3}	- 3rd harmonic*
Y_{n-2}	- 2nd harmonic*
Y_{n-1}	- 1st harmonic*

*These entries represent *negative* harmonics.

The following illustration represents this complex sequence.



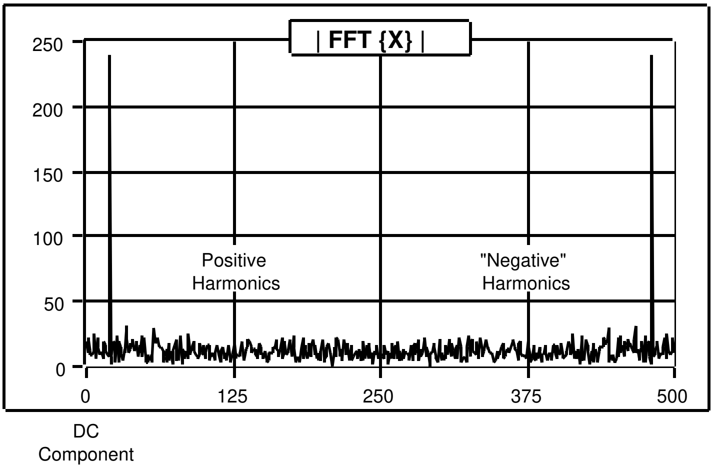
If n=7, let $k = \frac{n-1}{2}$. The following table shows the format of the complex output sequence Y.

Array Element	Interpretation
Y_0	DC component
Y_1	1st harmonic or fundamental
Y_2	2nd harmonic
Y_3	3rd harmonic
. . .	. . .

Array Element	Interpretation
Y_{k-1}	k^{th}-1 harmonic
Y_k	k^{th} harmonic
$Y_{k+1} = Y_{n-k} = Y_{-k}$	-k^{th} harmonic*
$Y_{k+2} = Y_{n-(k-1)} = Y_{-(k-1)}$	- $(k\text{-}1)^{th}$ harmonic*
. . .	. . .
Y_{n-3}	- 3^{rd} harmonic*
Y_{n-2}	- 2^{nd} harmonic*
Y_{n-1}	- 1^{st} harmonic*

*These entries represent *negative* harmonics.

The following illustration represents the preceding table.



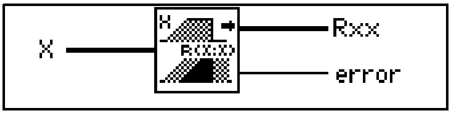
This format is an accepted standard in digital signal processing applications. It is convenient because it simplifies performing the inverse transform to obtain the final, processed result.

Signal Processing VI Descriptions

The following Signal Processing VIs are available.

AutoCorrelation

Computes the autocorrelation of the input sequence **X**.



The autocorrelation $R_{xx}(t)$ of a function $x(t)$ is defined as

$$R_{xx}(t) = x(t) \otimes x(t) = \int_{-\infty}^{\infty} x(\tau)x(t+\tau)dt \ ,$$

where the symbol ⊗ denotes correlation.

For the discrete implementation of this VI, let Y represent a sequence whose indexing can be negative, let n be the number of elements in the input sequence **X**, and assume that the indexed elements of **X** that lie outside its range are equal to zero,

$x_j = 0, \quad j < 0 \quad \text{or} \quad j \geq n.$

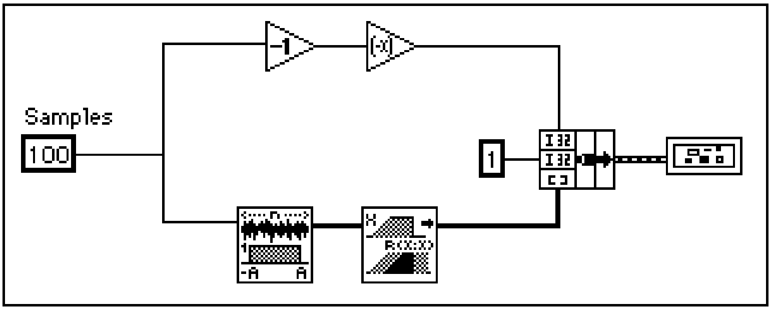
Then the VI obtains the elements of Y using

$$y_j = \sum_{k=0}^{n-1} x_k x_{j+k} \quad \text{for } j = -(n-1), -(n-2), \ldots, -2, -1, 0, 1, 2, \ldots, n-1.$$

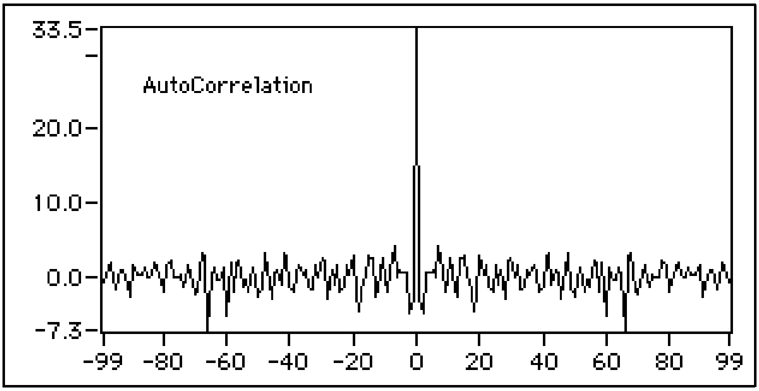
The elements of the output sequence **Rxx** are related to the elements in the sequence Y by:

$Rxx_i = y_{i-(n-1)}$ for $i = 0, 1, 2, \ldots, 2n-2$.

Notice that the number of elements in the output sequence **Rxx** is $2n$ - 1. Because you cannot use negative numbers to index LabVIEW arrays, the corresponding correlation value at $t = 0$ is the n^{th} element of the output sequence **Rxx**. Therefore, **Rxx** represents the correlation values that the VI shifted *n* times in indexing. The following block diagram shows one way to display the correct indexing for the autocorrelation function.

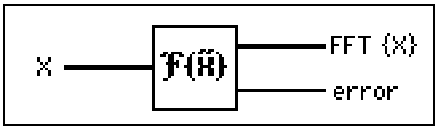


The following graph is the result of the preceding block diagram.



Complex FFT

Computes the Fourier transform of the input sequence **X**.



You can use this VI to perform an FFT on an array of complex numeric representations.

If *Y* represents the complex output sequence, then

$Y = F\{X\}.$

You also can use this VI to perform the following operations when **X** has one of the complex LabVIEW data types.

- The FFT of a complex-valued sequence *X*
- The DFT of a complex-valued sequence *X*

This VI first analyzes the input data, and based on this analysis, it calculates the Fourier transform of the data by executing one of the preceding options. All these routines take advantage of the concurrent processing capabilities of the CPU and FPU.

When the number of samples in the input sequence **X** is a valid power of 2,

$n = 2^m \qquad \text{for } m = 1, 2, 3,\ldots, 23,$

where *n* is the number of samples, the VI computes the fast Fourier transform by applying the split-radix algorithm. The largest complex FFT the VI can compute is 2^{23} = 8,388,608 (8M).

When the number of samples in the input sequence **X** is *not* a valid power of 2,

$n \neq 2^m \qquad \text{for } m = 1, 2, 3,\ldots, 23,$

where *n* is the number of samples, the VI computes the discrete Fourier transform by applying the chirp-z algorithm. The largest complex DFT that can be computed is 2^{22}-1 = 4,194,303 (4M - 1).

☞ **Note:** ***Because the VI performs the transform in place, advantages of the FFT include speed and memory efficiency. The size of the input sequence, however, must be a power of 2. The DFT can efficiently process any size sequence, but the DFT is slower than the FFT and uses more memory, because it must store intermediate results during processing.***

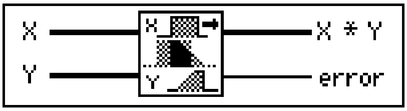
Let *Y* be the complex output sequence and *n* be the number of samples in it. Using equation (40-7), you can show that

$Y_{n-i} = Y_{-i}$

which means you can interpret the $(n\text{-}i)^{\text{th}}$ element of *Y* as the $\text{-}i^{\text{th}}$ element of the sequence, if it could be physically realized, which represents the negative i^{th} harmonic.

Convolution

Computes the convolution of the input sequences **X** and **Y**.



The convolution $h(t)$, of the signals $x(t)$ and $y(t)$ is defined as

$$h(t) = x(t)*y(t) = \int_{-\infty}^{\infty} x(\tau)y(t-\tau)d\tau$$

where the symbol $*$ denotes convolution.

For the discrete implementation of the convolution, let h represent the output sequence **X * Y**, let n be the number of elements in the input sequence **X**, and let m be the number of elements in the input sequence **Y**. Assuming that indexed elements of **X** and **Y** that lie outside their range are zero,

$x_i = 0, \quad i < 0 \quad \text{or} \quad i \geq n$

and

$y_j = 0, \quad j < 0 \quad \text{or} \quad j \geq m,$

then you obtain the elements of h using

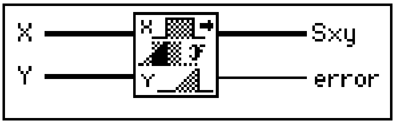
$$h_i = \sum_{k=0}^{n-1} x_k y_{i-k} \quad \text{for } i = 0, 1, 2, \ldots, \text{size-1},$$

size = $n + m$ - 1,

where size denotes the total number of elements in the output sequence **X * Y**.

Cross Power

Computes the cross power spectrum of the input sequences **X** and **Y**.



The cross power, $S_{xy}(f)$, of the signals $x(t)$ and $y(t)$ is defined as

$S_{xy}(f) = X^*(f)Y(f)$

where $X^*(f)$ is the complex conjugate of $X(f)$, $X(f) = F\{x(t)\}$, and $Y(f) = F\{y(t)\}$.

This VI uses the FFT or DFT routine to compute the cross power spectrum, which is given by

$$S_{xy} = \frac{1}{n^2}F^*\{X\}F\{Y\} \quad ,$$

where S_{xy} represents the complex output sequence **Sxy**, and n is the number of samples that can accommodate both input sequences **X** and **Y**.

The largest cross power that the VI can compute via the FFT is 2^{23} (8,388,608 or 8M).

When the number of samples in **X** and **Y** are equal and are a valid power of 2,

$n = m = 2^k$ for $k = 1, 2, 3, \ldots, 23$,

where n is the number of samples in **X**, and m is the number of samples in **Y**, the VI makes direct calls to the FFT routine to compute the complex, cross power sequence. This method is extremely efficient in both execution time and memory management because the VI performs the operations in place.

When the number of samples in **X** and **Y** are not equal,

$n \neq m$,

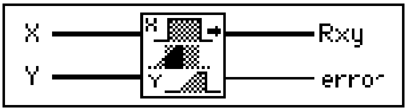
where n is the number of samples in **X**, and m is the number of samples in **Y**, the VI first resizes the smaller sequence by padding it with zeros to match the size of the larger sequence. If this size is a valid power of 2,

$\max(n,m) = 2^k$ for $k = 1, 2, 3, \ldots, 23$, the VI computes the cross power spectrum using the FFT; otherwise the VI uses the slower DFT to compute the cross power spectrum. Thus, the size of the complex output sequence is

size = max(n,m).

CrossCorrelation

Computes the cross correlation of the input sequences **X** and **Y**.



The cross correlation $R_{xy}(t)$ of the signals $x(t)$ and $y(t)$ is defined as

$$R_{xy}(t) = x(t) \otimes y(t) = \int_{-\infty}^{\infty} x(\tau)y(t+\tau)d\tau ,$$

where the symbol ⊗ denotes correlation.

For the discrete implementation of this VI, let h represent a sequence whose indexing can be negative, let n be the number of elements in the input sequence **X**, let m be the number of elements in the sequence **Y**, and assume that the indexed elements of **X** and **Y** that lie outside their range are equal to zero,

$x_j = 0, \quad j < 0 \quad \text{or} \quad j \geq n,$

and

$y_j = 0, \quad j < 0 \quad \text{or} \quad j \geq m.$

Then the VI obtains the elements of h using

$$h_j = \sum_{k=0}^{n-1} x_k y_{j+k} \quad \text{for } j = \text{-}(n\text{-}1), \text{-}(n\text{-}2), \ldots, \text{-}2, \text{-}1, 0, 1, 2, \ldots, m\text{-}1.$$

The elements of the output sequence **Rxy** are related to the elements in the sequence *h* by

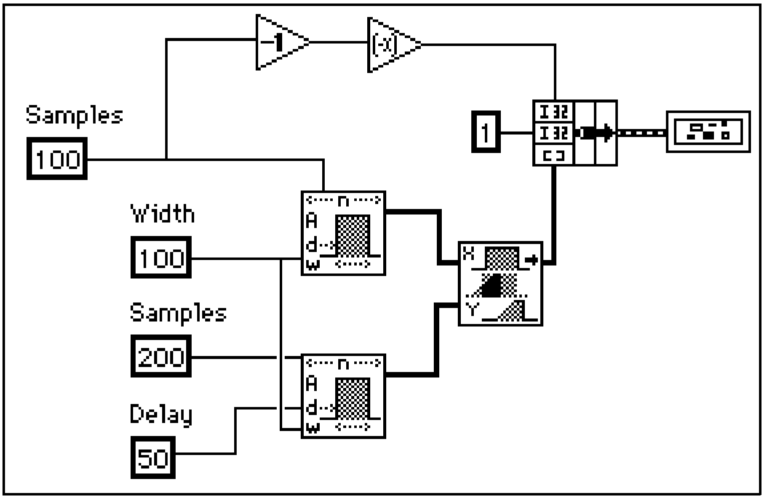
$Rxy_i = h_{i-(n-1)}$ for $i = 0, 1, 2, \ldots,$ size-1,

size = $n + m - 1$

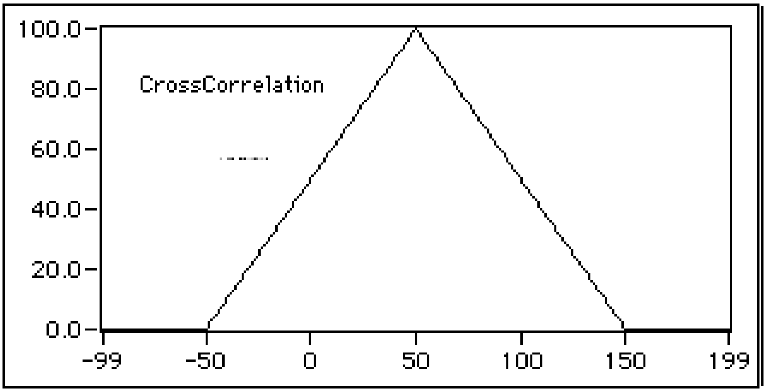
where size is the number of elements in the output sequence **Rxy**.

Because you cannot index LabVIEW arrays with negative numbers, the corresponding cross correlation value at $t = 0$ is the n^{th} element of the output sequence **Rxy**. Therefore, **Rxy** represents the correlation values that the VI shifted *n* times in indexing.

The following block diagram shows one way to index the CrossCorrelation VI.

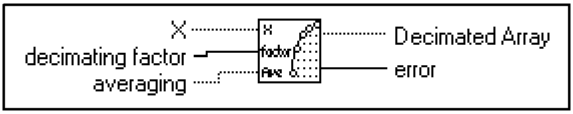


The following graph is the result of the preceding block diagram.



Decimate

Decimates the input sequence **X** by the **decimating factor** and the **averaging** binary control.



If *Y* represents the output sequence **Decimated Array**, the VI obtains the elements of the sequence *Y* using

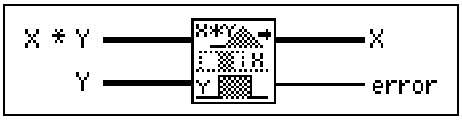
$$y_i = \begin{cases} x_{im} & \text{if ave is false} \\ \dfrac{1}{m}\sum_{k=0}^{m-1} x_{(im+k)} & \text{if ave is true} \end{cases} \qquad \text{for } i = 0, 1, 2, \ldots, \text{size-1}$$

$$\text{size} = \text{trunc}\left(\frac{n}{m}\right),$$

where *n* is the number of elements in **X**, *m* is the **decimating factor**, *ave* is the **averaging** option, and *size* is the number of elements in the output sequence **Decimated Array**.

Deconvolution

Computes the deconvolution of the input sequences **X * Y** and **Y**.



The VI can use Fourier identities to realize the convolution operation because

$x(t) * y(t) \Leftrightarrow X(f)\ Y(f)$

is a Fourier transform pair, where the symbol $*$ denotes convolution, and the deconvolution is the inverse of the convolution operation. If $h(t)$ is the signal resulting from the deconvolution of the signals $x(t)$ and $y(t)$, the VI obtains $h(t)$ using the equation

$$h(t) = F^{-1}\left(\frac{X(f)}{Y(f)}\right) ,$$

where $X(f)$ is the Fourier transform of $x(t)$, and $Y(f)$ is the Fourier transform of $y(t)$.

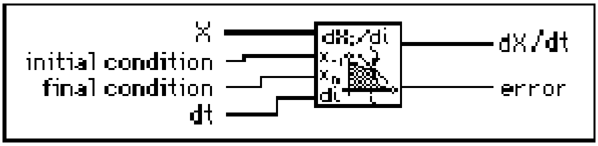
The VI performs the discrete implementation of the deconvolution using the following steps:

1. Compute the Fourier transform of the input sequence **X * Y**.
2. Compute the Fourier transform of the input sequence **Y**.
3. Divide the Fourier transform of **X * Y** by the Fourier transform of **Y**. Call the new sequence *H*.
4. Compute the inverse Fourier transform of *H* to obtain the deconvoluted sequence **X**.

☞ **Note:** ***The deconvolution operation is a numerically unstable operation, and it is not always possible to solve the system numerically. Computing the deconvolution via FFTs is perhaps the most stable generic algorithm not requiring sophisticated DSP techniques. However, it is not free of errors (for example, when there are zeros in the Fourier transform of the input sequence Y).***

Derivative x(t)

Performs a discrete differentiation of the sampled signal **X**.



The differentiation $f(t)$ of a function $F(t)$ is defined as

$$f(t) = \frac{d}{dt}F(t) .$$

Let Y represent the sampled output sequence **dX/dt**. The discrete implementation is given by

$$y_i = \frac{1}{2dt}(x_{i+1} - x_{i-1}) \quad \text{for } i = 0, 1, 2, \ldots, n\text{-}1,$$

where n is the number of samples in **x(t)**,

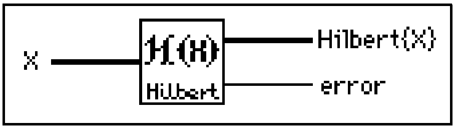
x_{-1} is specified by **initial condition** when $i = 0$, and

x_n is specified by **final condition** when $i = n$-1.

The **initial condition** and **final condition** minimize the error at the boundaries.

Fast Hilbert Transform

Computes the fast Hilbert transform of the input sequence **X**.



The Hilbert transform of a function $x(t)$ is defined as

$$h(t) = H\{x(t)\} = -\frac{1}{\pi}\int_{-\infty}^{\infty} \frac{x(\tau)}{t-\tau}d\tau .$$

Using Fourier identities, you can show the Fourier transform of the Hilbert transform of $x(t)$ is

$h(t) \Leftrightarrow H(f) = -j\ \mathrm{sgn}(f)\ X(f)$

where $x(t) \Leftrightarrow X(f)$ is a Fourier transform pair and

$$\mathrm{sgn}(f) = \begin{cases} 1 & f > 0 \\ 0 & f = 0 \\ -1 & f < 0 \end{cases}$$

The VI completes the following steps to perform the discrete implementation of the Hilbert transform with the aid of the FFT routines based upon the $h(t) \Leftrightarrow H(f)$ Fourier transform pair (refer to the output format of the FFT VI for more information):

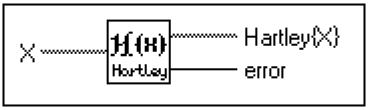
1. Fourier transform the input sequence **X**: $Y = F\{X\}$.
2. Set the DC component to zero: $Y_0 = 0$.
3. If the sequence Y is an even size, set the Nyquist component to zero: $Y_{\mathrm{Nyq}} = 0$.
4. Multiply the positive harmonics by $-j$.
5. Multiply the negative harmonics by j. Call the new sequence H, which is of the form $H_k = -j\ \mathrm{sgn}(k)\ Y_k$.
6. Inverse Fourier transform H to obtain the Hilbert transform of **X**.

You use the Hilbert transform to extract instantaneous phase information, obtain the envelope of an oscillating signal, obtain single-sideband spectra, detect echoes, and reduce sampling rates.

☞ **Note:** ***Because the VI sets the DC and Nyquist components to zero when the number of elements in the input sequence is even, you cannot always recover the original signal with an inverse Hilbert transform. The Hilbert transform works well with bandpass limited signals, which exclude the DC and the Nyquist components.***

FHT

Computes the fast Hartley transform (FHT) of the input sequence **X**.



The Hartley transform of a function $x(t)$ is defined as

$$X(f) = \int_{-\infty}^{\infty} x(t)\mathrm{cas}(2\pi f t)dt$$

where $\mathrm{cas}(x) = \cos(x) + \sin(x)$.

If Y represents the output sequence **Hartley{X}** obtained via the FHT, then Y is obtained through the discrete implementation of the Hartley integral:

$$Y_k = \sum_{i=0}^{n-1} X_i \mathrm{cas}\left(\frac{2\pi ik}{n}\right), \text{ for } k = 0, 1, 2, \ldots, n\text{-}1.$$

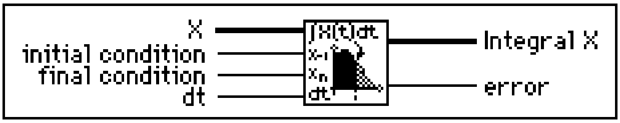
where n is the number of elements in **X**.

FHT maps real-valued sequences into real-valued frequency domain sequences. You can use it instead of the Fourier transform to convolve signals, deconvolve signals, correlate signals, and find the power spectrum. You can also derive the Fourier transform from the Hartley transform.

When the sequences to be processed are real-valued sequences, the Fourier transform produces complex-valued sequences in which half of the information is redundant. The advantage of using the FHT instead of the FFT transform is that the FHT uses half the memory to produce the same information the FFT produces. Further, the FHT is calculated in place and is as efficient as the FFT. The disadvantage of the FHT is that the size of the input sequence must be a valid power of 2.

Integral x(t)

Performs the discrete integration of the sampled signal **X**.



The integral $F(t)$ of a function $f(t)$ is defined as

$$F(t) = \int f(t)dt$$

Let *Y* represent the sampled output sequence **Integral X**. The VI obtains the elements of *Y* using

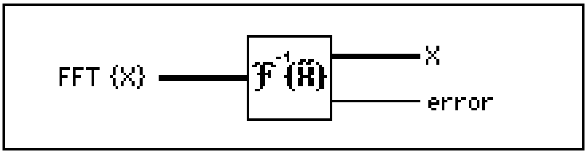
$$y_i = \frac{1}{6}\sum_{j=0}^{i}(x_{j-1} + 4x_j + x_{j+1})dt \quad \text{for } i = 0, 1, 2, \ldots, n\text{-}1,$$

where *n* is the number of elements in **X**, x_{-1} is specified by **initial condition** when $i = 0$, and x_n is specified by **final condition** when $i = n$-1.

The **initial condition** and **final condition** minimize the overall error by increasing the accuracy at the boundaries, especially when the number of samples is small. Determining boundary conditions before the fact enhances accuracy.

Inverse Complex FFT

Computes the inverse Fourier transform of the complex input sequence **FFT {X**}.



You can use this VI to perform an inverse FFT on an array of one of the LabVIEW complex numeric representations.

If *Y* represents the output sequence, then

$Y = F^{-1}\{X\}$.

You can use this VI to perform the following operations when **FFT {X**} has one of the complex LabVIEW data types:

- The inverse FFT of a complex-valued sequence *X*
- The inverse DFT of a complex-valued sequence *X*

This FFT VI first analyzes the input data and, based on this analysis, inverse Fourier transforms the data by executing one of the preceding options. All these routines take advantage of the concurrent processing capabilities of the CPU and FPU.

When the number of samples in the input sequence *X* is a valid power of 2,

$n = 2^m \qquad \text{for } m = 1, 2, 3, \ldots, 23,$

where *n* is the number of samples, the VI computes the inverse FFT by applying the split-radix algorithm. The longest sequence with an inverse complex FFT that the VI can compute is 2^{23}=8,388,608 (8M).

When the number of samples in the input sequence *X* is *not* a valid power of 2,

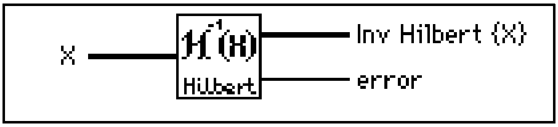
$n \neq 2^m$ for $m = 1, 2, 3, \ldots, 23$,

where *n* is the number of samples, the VI computes the inverse DFT by applying the chirp-z algorithm. The longest sequence with an inverse complex DFT that the VI can compute is 2^{22}-1 (4,194,303 or 4M - 1).

Note: ***Because the VI performs the transform in place, advantages of the FFT include speed and memory efficiency. The size of the input sequence, however, must be a power of 2. The DFT can efficiently process any size sequence, but the DFT is slower than the FFT and uses more memory, because it must store intermediate results during processing.***

Inverse Fast Hilbert Transform

Computes the inverse fast Hilbert transform of the input sequence **X**.



The inverse Hilbert transform of a function *h*(*t*) is defined as

$$h(t) = H^{-1}\{h(t)\} = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{h(\tau)}{t-\tau}d\tau \quad .$$

Using the definition of the Hilbert transform

$$h(t) = H\{x(t)\} = -\frac{1}{\pi}\int_{-\infty}^{\infty}\frac{x(\tau)}{t-\tau}d\tau \quad ,$$

you obtain the inverse Hilbert transform by negating the forward Hilbert transform

$x(t) = H^{-1}\{h(t)\} = - H\{h(t)\}$.

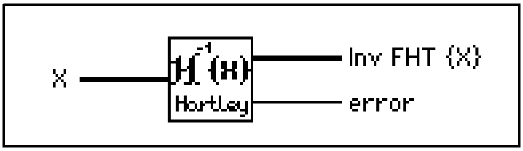
The VI completes the following steps to perform the discrete implementation of the inverse Hilbert transform with the aid of the Hilbert transform.

1. Hilbert transform the input sequence **X**: $Y = H\{X\}$.
2. Negate Y to obtain the inverse Hilbert transform: $H^{-1}\{X\} = -Y$.

For more information on the algorithm this VI uses, refer to the description of the *Fast Hilbert Transform* section in this chapter.

Inverse FHT

Computes the inverse fast Hartley transform of the input sequence **X**.



The inverse Hartley transform of a function $X(f)$ is defined as

$$x(t) = \int_{-\infty}^{\infty} X(f)\operatorname{cas}(2\pi f t)\,df$$

where cas(x) = cos(x) + sin(x).

If Y represents the output sequence **Inv FHT {X}**, the VI calculates Y through the discrete implementation of the inverse Hartley integral:

$$Y_k = \frac{1}{n}\sum_{i=0}^{n-1} X_i \operatorname{cas}\frac{2\pi i k}{n}\text{, for } k = 0, 1, 2, \ldots, n\text{-}1.$$

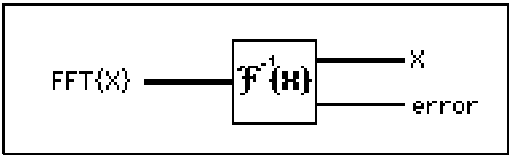
where n is the number of elements in **X**.

The inverse Hartley transform maps real-valued frequency sequences into real-valued sequences. You can use it instead of the inverse Fourier transform to convolve, deconvolve, and correlate signals. You can also derive the Fourier transform from the Hartley transform.

See the *FHT* section for a comparison of the Fourier and Hartley transforms.

Inverse Real FFT

Computes the Inverse Real Fast Fourier Transform (FFT) or the Inverse Real Discrete Fourier Transform (DFT) of the input sequence **FFT{X}**.



The input sequence is complex-valued. This VI automatically determines the following options:

- Inverse Real FFT of a complex-valued sequence if the size is a power of 2.
- Inverse Real DFT of a complex-valued sequence if the size is not a power of 2.

This VI executes inverse FFT routines if the size of the input sequence is a valid power of 2:

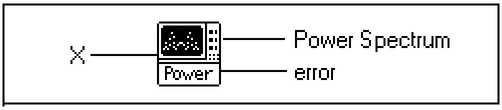
size = 2^m, $m = 1, 2, \ldots, 23$.

If the size of the input sequence is not a power of 2, this VI calls an efficient Inverse DFT routine.

The output sequence **X** = Inverse Real FFT [**FFT{X}**] is real and it returns in one real array.

Power Spectrum

Computes the power spectrum of the input sequence **X**.



The **Power Spectrum** $S_{xx}(f)$ of a function $x(t)$ is defined as

$$S_{xx}(f) = X^*(f)X(f) = |\, X(f) \,|^2$$

where $X(f) = F\{x(t)\}$, and $X^*(f)$ is the complex conjugate of $X(f)$.

This VI uses the FFT and DFT routines to compute the power spectrum, which is given by

$$S_{xx} = \frac{1}{n^2}|F\{\mathbf{X}\}|^2,$$

where S_{xx} represents the output sequence **Power Spectrum**, and n is the number of samples in the input sequence **X**.

When the number of samples, n, in the input sequence **X** is a valid power of 2,

$n = 2^m$ for $m = 1, 2, 3, \ldots, 23$,

this VI computes the FFT of a real-valued sequence using the split-radix algorithm and efficiently scales the magnitude square. The largest power spectrum the VI can compute using the FFT is 2^{23} (8,388,608 or 8M).

When the number of samples in the input sequence *X* is *not* a valid power of 2,

$n \neq 2^m$ for $m = 1, 2, 3,\ldots, 23$,

where n is the number of samples, this VI computes the discrete Fourier transform of a real-valued sequence using the chirp-z algorithm and scales the magnitude square. The largest power spectrum the VI can compute using the fast DFT is 2^{22}-1 (4,194,303 or 4M - 1).

The FFT computation of the power spectrum is time and memory efficient because the transform is real and done in the same space. However, the size of the input sequence must be exactly a power of 2. The DFT version efficiently computes the power spectrum of any size sequence. The DFT version is slower than the FFT version, uses more memory, and is not as efficient in scaling.

Let Y be the Fourier transform of the input sequence **X** and let n be the number of samples in it. Using equation (40-7), you can show that

$$|Y_{n-i}|^2 = |Y_{-i}|^2.$$

You can interpret the power in the (n-i)th element of Y as the power in the -ith element of the sequence, which represents the power in the *negative* ith harmonic. You can find the total power for the ith harmonic (DC and Nyquist component not included) using

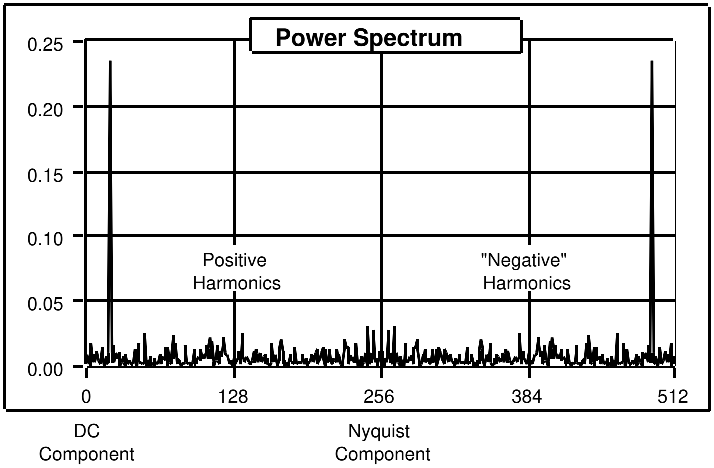
$$\text{Power in } i^{th} \text{ harmonic} = 2|Y_i|^2 = |Y_i|^2 + |Y_{n-i}|^2, \quad 0 < i < \frac{n}{2}.$$

The total power in the DC and Nyquist components are $|Y_0|^2$ and $|Y_{n/2}|^2$, respectively.

If n is even, let $k = \frac{n}{2}$. The following table shows the format of the output sequence Sxx corresponding to the power spectrum.

Array Element	Interpretation
Sxx_0	Power in DC component
$Sxx_1 = Sxx_{(n\text{-}1)}$	Power in 1st harmonic or fundamental
$Sxx_2 = Sxx_{(n\text{-}2)}$	Power in 2nd harmonic
$Sxx_3 = Sxx_{(n\text{-}3)}$	Power in 3rd harmonic
. . .	. . .
$Sxx_{k\text{-}2} = Sxx_{n\text{-}(k\text{-}2)}$	Power in (k-2)th harmonic
$Sxx_{k\text{-}1} = Sxx_{n\text{-}(k\text{-}1)}$	Power in (k-1)th harmonic
Sxx_{k}	Power in Nyquist harmonic

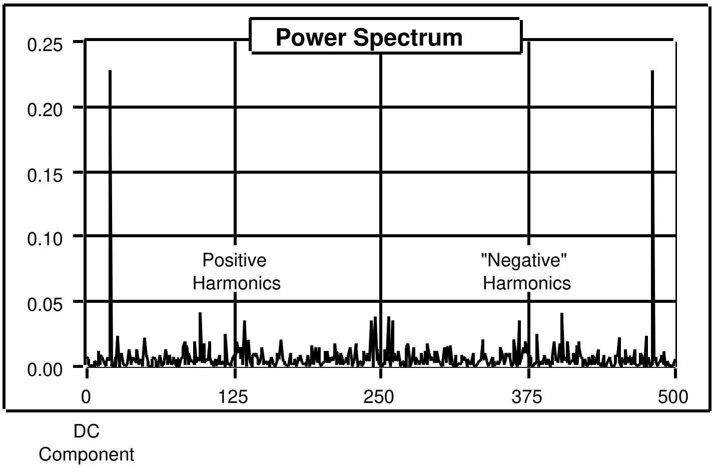
The following illustration represents the preceding table information.



If n is odd, let $k = \frac{n-1}{2}$. The following table shows the format of the output sequence Sxx corresponding to the power spectrum.

Array Element	Interpretation
Sxx_0	Power in DC component
$Sxx_1 = Sxx_{(n\text{-}1)}$	Power in 1st harmonic or fundamental
$Sxx_2 = Sxx_{(n\text{-}2)}$	Power in 2nd harmonic
$Sxx_3 = Sxx_{(n\text{-}3)}$	Power in 3rd harmonic
. . .	. . .
$Sxx_{k\text{-}2} = Sxx_{n\text{-}(k\text{-}2)}$	Power in (k-2)th harmonic
$Sxx_{k\text{-}1} = Sxx_{n\text{-}(k\text{-}1)}$	Power in (k-1)th harmonic
$Sxx_k = Sxx_{n\text{-}k}$	Power in kth harmonic

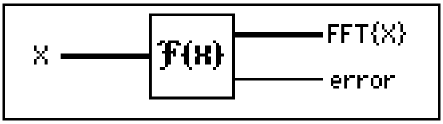
The following illustration represents the preceding table information.



The format described in the preceding tables is an accepted standard in digital signal processing applications.

Real FFT

Computes the Real Fast Fourier Transform (FFT) or the Real Discrete Fourier Transform (DFT) of the input sequence **X**.



The input sequence is real-valued. The Real FFT VI automatically determines the options, which are the following:

- FFT of a real-valued sequence
- DFT of a real-valued sequence

The Real FFT VI executes FFT routines if the size of the input sequence is a valid power of 2:

size = 2^m, m = 1, 2,..., 23.

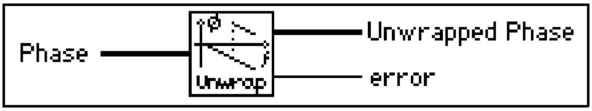
If the size of the input sequence is not a power of 2, the Real FFT VI calls an efficient Real DFT routine.

The output sequence Y = Real FFT[**X**] is complex and returns in one complex array:

$Y = Y\mathrm{Re} + jY\mathrm{Im}$

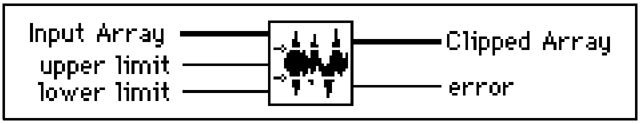
Unwrap Phase

Unwraps the **Phase** array by eliminating discontinuities whose absolute values exceed π.



Y[i] = Clip {X[i]}

Clips the elements of **Input Array** to within the bounds specified by **upper limit** and **lower limit**.



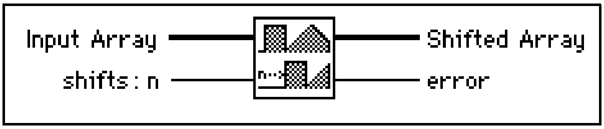
Let the sequence *Y* represent the output sequence **Clipped Array**; then the elements of *Y* are related to the elements of **Input Array** by

$$y_i = \begin{cases} a & x_i > a \\ x_i & b \le x_i \le a \\ b & x_i < b \end{cases} \quad \text{for } i = 0, 1, 2, \ldots, n\text{-}1,$$

where *n* is the number of elements in **Input Array**, *a* is the **upper limit**, and *b* is the **lower limit**.

Y[i] = X[i-n]

Shifts the elements in the **Input Array** by the specified number of shifts.



Let the sequence *Y* represent the output sequence **Shifted Array**; then the elements of *Y* are related to the elements of **X** by

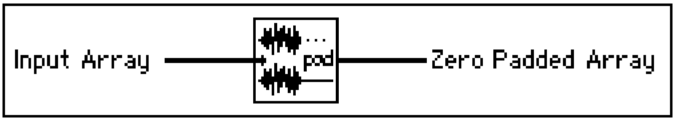
$$y_i = \begin{cases} x_{i-shifts} & \text{if } 0 \le i - shifts < n \quad \text{for i = 0, 1, 2, . . ., n-1 ,} \\ 0 \quad \text{elsewhere} & \end{cases}$$

where *n* is the number of elements in **Input Array**.

Note: ***This VI does not rotate the elements in the array. The VI disposes of the elements of the input sequence shifted outside the range, and you cannot recover them by shifting the array in the opposite direction.***

Zero Padder

Resizes the input sequence **Input Array** to the next higher valid power of 2, sets the new trailing elements of the sequence to zero, and leaves the first *n* elements unchanged, where *n* is the number of samples in the input sequence.



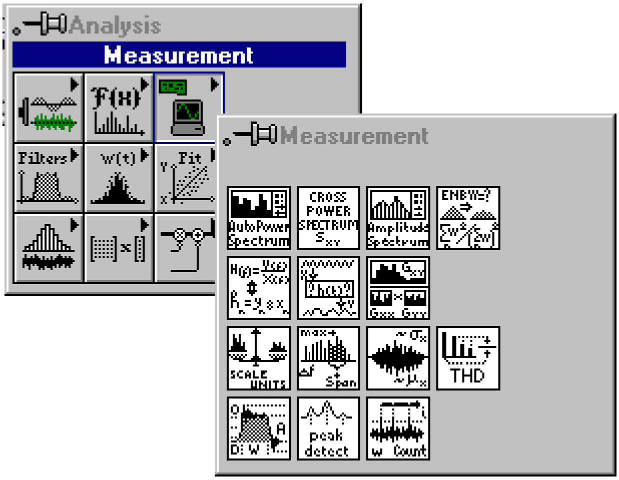
This VI is useful when the size of the acquired data buffers is not a power of 2, and you want to take advantage of fast processing algorithms in the analysis VIs. These algorithms include Fourier transforms, power spectrum, and FHTs, which are extremely efficient for buffer sizes that are a power of 2.

Chapter 41

Analysis Measurement VIs

This chapter describes the measurement VIs, which are streamlined to perform DFT-based and FFT-based analysis with signal acquisition for frequency measurement applications as seen in typical frequency measurement instruments, such as dynamic signal analyzers.

To access the Analysis Measurement palette, select **Function»Analysis»Measurement**. The following illustration shows the options that are available on the Measurement palette.



For examples of how to use the measurement VIs, see the examples using data acquisition located in `examples\analysis\measure\daqmeas.llb` (Windows and Macintosh) and using simulated signals in `examples\analysis\measure\measxmpl.llb`.

Introduction to Measurement VIs

Several measurement VIs perform commonly used time domain to frequency-domain transformations such as amplitude and phase spectrum, signal power spectrum, network transfer function, and so on. Other measurement VIs interact with VIs that perform such functions as scaled time domain windowing and power and frequency estimation.

You can use the measurement VIs for the following applications:

- Spectrum analysis applications
  - Amplitude and phase spectrum
  - Power spectrum
  - Scaled time domain window
  - Power and frequency estimate
  - Harmonic analysis and total harmonic distortion (THD) measurements
- Network (frequency response) and dual channel analysis applications
  - Transfer function
  - Impulse response function
  - Network functions (including coherence)
  - Cross power spectrum

The DFT, FFT, and power spectrum are useful for measuring the frequency content of stationary or transient signals. The FFT provides the average frequency content of the signal over the entire time that the signal was acquired. For this reason, you use the FFT mostly for stationary signal analysis (when the signal is not significantly changing in frequency content over the time that the signal is acquired), or when you want only the average energy at each frequency line. A large class of measurement problems fall in this category. For measuring frequency information that changes during the acquisition, you should use joint time-frequency analysis VIs, such as the Gabor Spectrogram.

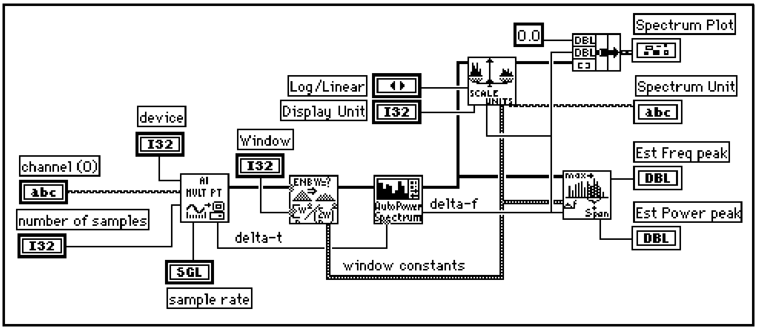
The measurement VIs are built on top of the signal processing VIs and have the following characteristics, which model the behavior of traditional, benchtop frequency analysis instruments.

- Real-world, time-domain signal input is assumed.

- Outputs are in magnitude and phase, scaled, and in units where appropriate, ready for immediate graphing.
- Single-sided spectrums from DC to $\frac{\text{Sampling Frequency}}{2}$.
- Sampling period to frequency interval conversion for graphing with appropriate X axis units (in Hz).
- Corrections for the windows being used are applied where appropriate.
- Windows are scaled so that each window gives the same peak spectrum amplitude result within its amplitude accuracy constraints.

Views power or amplitude spectrums in various unit formats, including decibels and spectral density units, such as V^2/Hz, $V/\sqrt{Hz}$, and so on.

In general, you can directly connect the measurement VIs to the output of data acquisition VIs and to graphs through the axis cluster, as the following spectrum analyzer diagram shows.



The measurement examples include the following:

- Amplitude Spectrum Example
- Simulated Dynamic Signal Analysis Example
- Total Harmonic Distortion (THD) Example

(Windows and Macintosh) You can use the following examples with National Instruments hardware.

- Simple Spectrum Analyzer and Spectrum Analyzer–Both work with any analog input hardware (use dynamic signal acquisition hardware for good quality measurements).

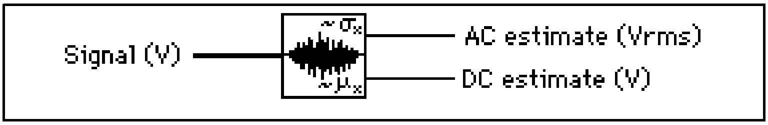
- Dynamic Signal Analyzer and Network Analyzer–Both work with dynamic signal acquisition hardware. The Network Analyzer requires the AT-DSP2200 board.

Measurement VI Descriptions

The following Measurement VIs are available.

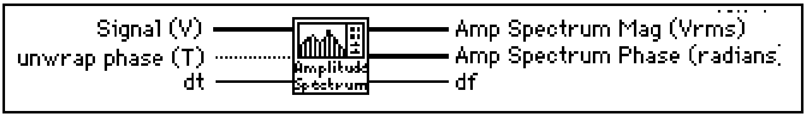
AC & DC Estimator

Computes an estimation of the AC and DC levels of the input signal.



Amplitude and Phase Spectrum

Computes the single-sided, scaled amplitude spectrum magnitude and phase of a real time-domain signal.



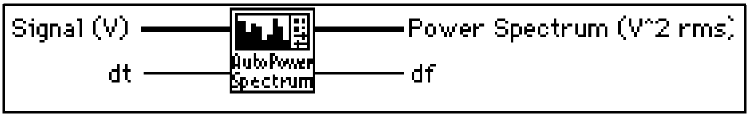
The VI computes the amplitude spectrum as

$$\frac{\text{FFT(Signal)}}{N}$$

where N is the number of points in the **Signal** array. The VI then converts the amplitude spectrum to single-sided rms magnitude and phase spectra.

Auto Power Spectrum

Computes the single-sided, scaled, auto power spectrum of a time-domain signal.



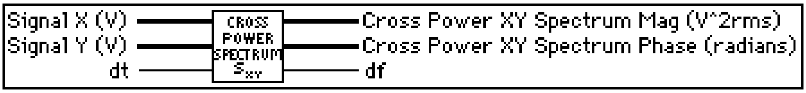
This VI computes the power spectrum as

$$\frac{\text{FFT*(Signal) x FFT(Signal)}}{N^2}$$

where *N* is the number of points in the **Signal** array and * denotes complex conjugate. The VI then converts the power spectrum into a single-sided power spectrum result.

Cross Power Spectrum

Computes the single-sided, scaled, cross power spectrum of two real-time signals. The cross power spectrum gives the product of the amplitude of the signals X and Y and the difference between their phases (phase of Y minus phase of X).


Signal X (V)
Signal Y (V)
dt
CROSS POWER SPECTRUM S_XY
Cross Power XY Spectrum Mag (V^2rms)
Cross Power XY Spectrum Phase (radians)
df

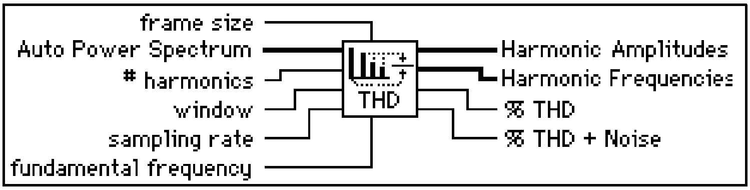
This VI computes the cross power spectrum as

$$\frac{\text{FFT*(Signal X) x FFT(Signal Y)}}{N^2}$$

where *N* is the number of points in **Signal X** or **Signal Y** arrays. The VI then converts the cross power spectrum to single-sided magnitude and phase spectra.

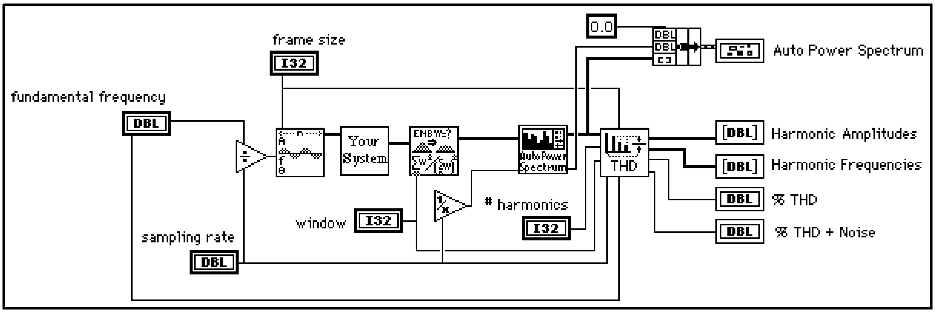
Harmonic Analyzer

Finds the fundamental and harmonic components (amplitude and frequency) present in the input **Auto Power Spectrum**, and computes the percent of total harmonic distortion (**%THD)** and the total harmonic distortion plus noise (**%THD + Noise**).



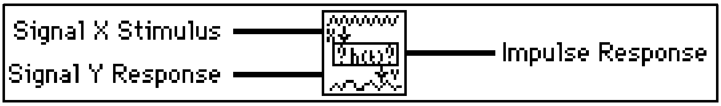
You must pass the windowed, auto power spectrum of your signal to this VI for it to function correctly. You should pass your time-domain signal through the scaled time domain window and then through the Auto Power Spectrum, connecting the Auto Power Spectrum output to this VI.

The following illustration shows an example of using this VI.



Impulse Response Function

Computes the impulse response of a network based on real signals X (**Signal X Stimulus**) and Y (**Signal Y Response**).



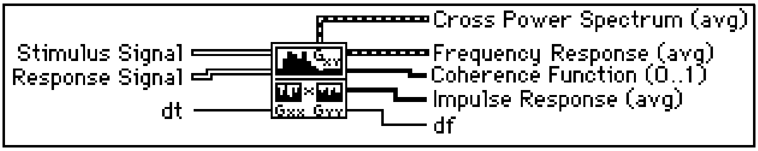
The **Impulse Response** is in the time domain, so you do not need to convert time units to frequency units. The **Impulse Response** is the inverse transform of the transfer function.

This VI computes **Impulse Response** as

$$\text{Inverse FFT}\left[\frac{\text{Cross Power(Stimulus, Response)}}{\text{Power Spectrum(Stimulus)}}\right].$$

Network Functions (avg)

Computes several network response functions of two, real time-domain signals X (**Stimulus Signal**) and Y (**Response Signal**).



The signals X (**Stimulus Signal**) and Y (**Response Signal**) include coherence, averaged cross power spectrum magnitude and phase, averaged transfer function (**Frequency Response**), and averaged **Impulse Response**.

You usually compute these functions on the stimulus and response signals from a network under test. The coherence function shows the frequency content of the **Response Signal** Y due to **Stimulus Signal** X and measures the validity of the network frequency response measurement.

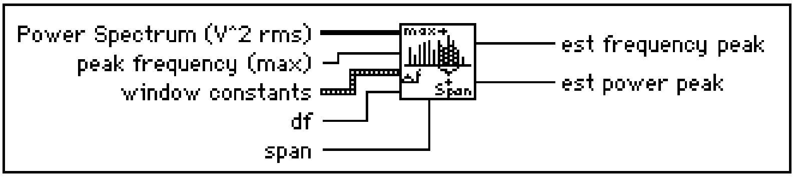
You can use this VI to measure the coherence between any two signals. The VI averages multiple stimulus and response signals to get valid coherence measurements. **Cross Power Spectrum** and **Impulse Response** are the rms averaged versions of the similarly named VIs. **Frequency Response** is the rms averaged version of the frequency response outputs of the Transfer Function VI.

Peak Detector

For information on this VI, see Chapter 48, *Analysis Additional Numerical Method VIs*, in this manual.

Power & Frequency Estimate

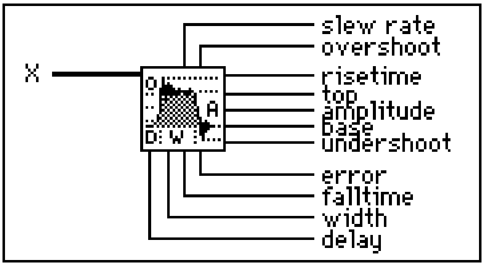
Computes the estimated power and frequency around a peak in the power spectrum of a time-domain signal.



With this VI, you can achieve good frequency estimates for measured frequencies that lie between frequency lines on the spectrum. The VI makes corrections for the window function you use.

Pulse Parameters

Analyzes the input sequence **X** for a pulse pattern and determines the best set of pulse parameters that describes the pulse.



The waveform-related parameters are **slew rate**, **overshoot**, topline (**top**), **amplitude**, baseline (**base**), and **undershoot**. The time-related parameters are **risetime**, **falltime**, **width** (duration), and **delay**.

This VI completes the following steps to calculate the output parameters:

1. Find the maximum and minimum values in the input sequence **X**.
2. Generate the histogram of the pulse with 1% range resolution.
3. Determine the upper and lower modes to establish the **top** and **base** values.
4. Find the **overshoot**, **amplitude**, and **undershoot** from **top**, **base**, maximum, and minimum values.
5. Scan **X** and determine the **slew rate**, **risetime**, **falltime**, **width**, and **delay**.

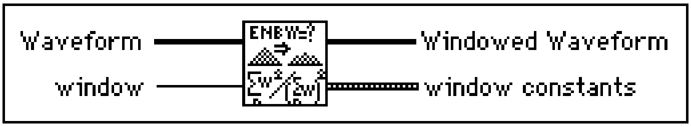
The VI interpolates **width** and **delay** to obtain a more accurate result not only of **width** and **delay,** but also of **slew rate**, **risetime**, and **falltime**.

If **X** contains a train of pulses, the VI uses the train to determine **overshoot**, **top**, **amplitude**, **base**, and **undershoot**, but uses only the first pulse in the train to establish **slew rate**, **risetime**, **falltime**, **width**, **and delay**.

Note: ***Because pulses commonly occur in the negative direction, this VI can discriminate between positive and negative pulses and can analyze the* X *sequence correctly. You do not need to process the sequence before analyzing it.***

Scaled Time Domain Window

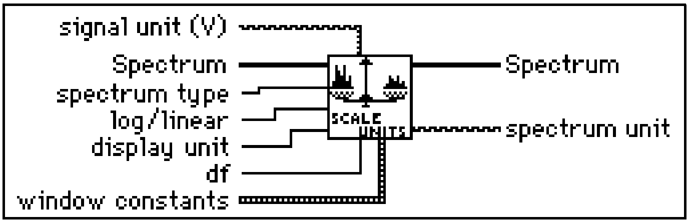
Applies the selected window to the time-domain signal.



The VI scales the result so that when the power or amplitude spectrum of the **Windowed Waveform** is computed, all windows provide the same level within the accuracy constraints of the window. This VI also returns important **Window Constants** for the selected window. These constants are useful when you use VIs that perform computations on the power spectrum, such as the Power & Frequency Estimate and Spectrum Unit Conversion VIs.

Spectrum Unit Conversion

Converts either the power, amplitude, or gain (amplitude ratio) spectrum to alternate formats including Log (decibel and dbm) and spectral density.

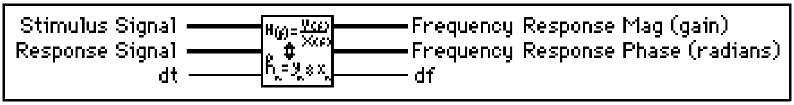


Threshold Peak Detector

For information on the this VI, see Chapter 48, *Analysis Additional Numerical Method VIs*, of this manual.

Transfer Function

Computes the transfer function (also known as the frequency response) from the time-domain **Stimulus Signal** and **Response Signal** from a network under test.


Stimulus Signal
Response Signal
dt
Frequency Response Mag (gain)
Frequency Response Phase (radians)
df

This VI computes the transfer function of a system based on the real signals X (**Stimulus Signal**) and Y (**Response Signal**). The output is the amplitude gain of the network, which is unitless.

The VI computer frequency response is:

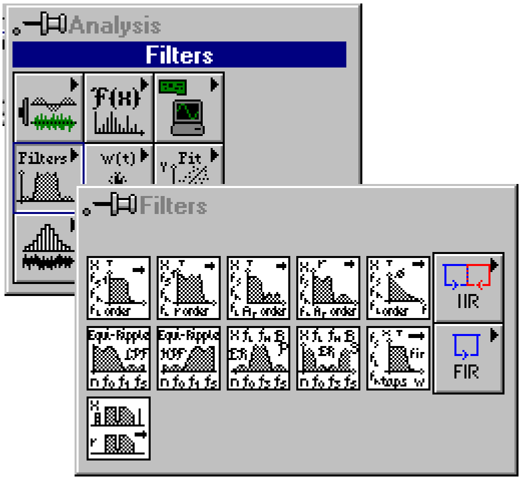
$$\frac{\text{Cross Power(Stimulus, Response)}}{\text{Power Spectrum(Stimulus)}}.$$

Chapter 42

Analysis Filter VIs

This chapter contains a brief discussion of digital filter theory and describes the VIs that implement IIR, FIR, and nonlinear filters.

To access the Analysis Filter palette, select **Function»Analysis»Filters**. The following illustration shows the options that are available on the Filter palette.



For examples of how to use the filter VIs, see the examples located in `examples\analysis\fltrxmpl.llb`.

Introduction to Digital Filtering Functions

Analog filter design is one of the most important areas of electronic design. Although analog filter design books featuring simple, tested filter designs exist, filter design is often reserved for specialists because it requires advanced mathematical knowledge and understanding of the processes involved in the system affecting the filter.

Modern sampling and digital signal processing tools have made it possible to replace analog filters with digital filters in applications that require flexibility and programmability. These applications include audio, telecommunications, geophysics, and medical monitoring.

Digital filters have the following advantages over their analog counterparts:

- They are software programmable.
- They are stable and predictable.
- They do not drift with temperature or humidity or require precision components.
- They have a superior performance-to-cost ratio.

You can use digital filters in LabVIEW to control parameters such as filter order, cutoff frequencies, amount of ripple, and stopband attenuation.

The digital filter VIs described in this section follow the virtual instrument philosophy. The VIs handle all the design issues, computations, memory management, and actual data filtering internally, and are transparent to the user. You do not have to be an expert in digital filters or digital filter theory to process the data.

The following discussion of sampling theory is intended to give you a better understanding of the filter parameters and how they relate to the input parameters.

The sampling theorem states that you can reconstruct a continuous-time signal from discrete, equally spaced samples if the sampling frequency is at least twice that of the highest frequency in the time signal. Assume you can sample the time signal of interest at Δt equally spaced intervals without losing information. The Δt parameter is the sampling interval.

You can obtain the sampling rate or sampling frequency f_s from the sampling interval

$$f_s = \frac{1}{\Delta t},$$

which means that, according to the sampling theorem, the highest frequency that the digital system can process is

$$f_{Nyq} = \frac{f_s}{2}.$$

The highest frequency the system can process is known as the Nyquist frequency. This also applies to digital filters. For example, if your sampling interval is

$\Delta t = 0.001$ sec,

then the sampling frequency is

$f_s = 1000$ Hz,

and the highest frequency that the system can process is

$f_{Nyq} = 500$ Hz.

The following types of filtering operations are based upon filter design techniques:

- Smoothing windows
- Infinite impulse response (IIR) or recursive digital filters
- Finite impulse response (FIR) or nonrecursive digital filters
- Nonlinear filters

The rest of this chapter presents a brief theoretical background on the IIR, FIR, and nonlinear techniques and discusses the digital filter VIs corresponding to each technique. Refer to Chapter 43, *Window VIs*, for information about the VIs that implement smoothing windows.

Infinite Impulse Response Filters

Infinite impulse response filters (IIR) are digital filters with impulse responses that can theoretically be infinite in length (duration). The general difference equation characterizing IIR filters is

$$y_i = \frac{1}{a_0}\left(\sum_{j=0}^{N_b-1} b_j x_{i-j} - \sum_{k=1}^{N_a-1} a_k y_{i-k}\right) \tag{42-1}$$

where N_b is the number of *forward* coefficients (b_j) and N_a is the number of *reverse* coefficients (a_k).

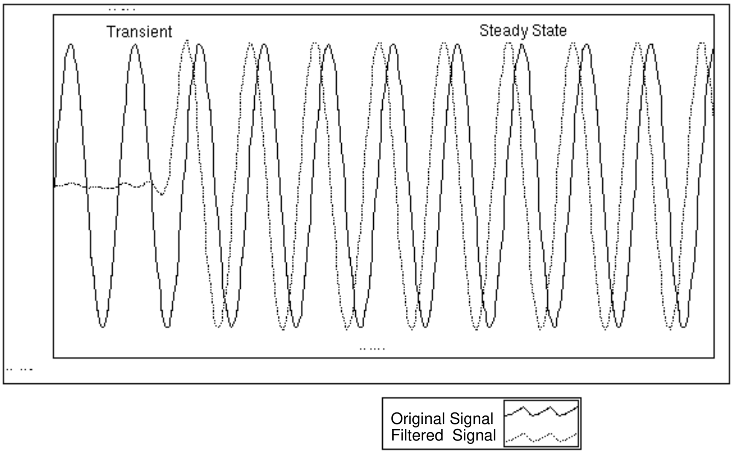
In most IIR filter designs (and in all of the LabVIEW IIR filters), coefficient a_0 is 1. The output sample at the present sample index i is the sum of scaled present and past inputs (x_i and x_{i-j} when ≠ 0) and scaled past outputs (y_{i-k}). Because of this, IIR filters are also known as recursive filters or autoregressive moving-average (ARMA) filters.

The response of the general IIR filter to an impulse ($x_0 = 1$ and $x_i = 0$ for all $i \neq 0$) is called the impulse response of the filter. The impulse response of the filter described by equation (42-1) is indeed of infinite length for nonzero coefficients. In practical filter applications, however, the impulse response of stable IIR filters decays to near zero in a finite number of samples.

IIR filters in LabVIEW contain the following properties:

- Negative indices resulting from equation (42-1) are assumed to be zero the first time you call the VI.
- Because the initial filter state is assumed to be zero (negative indices), a transient proportional to the filter order occurs before the filter reaches a steady state. The duration of the transient response, or delay, for lowpass and highpass filters is equal to the filter order.
- Delay = order.
- The duration of the transient response for bandpass and bandstop filters is twice the filter order
- Delay = 2 * order.

You can eliminate this transient response on successive calls by enabling state memory. To enable state memory, set the **init/cont** control of the VI to TRUE (continuous filtering).



The number of elements in the filtered sequence equals the number of elements in the input sequence.

The filter retains the internal filter state values when the filtering completes.

The advantage of digital IIR filters over finite impulse response (FIR) filters is that IIR filters usually require fewer coefficients to perform similar filtering operations. Thus, IIR filters execute much faster and do not require extra memory, because they execute in place.

The disadvantage of IIR filters is that the phase response is nonlinear. If the application does not require phase information, such as simple signal monitoring, IIR filters may be appropriate. You should use FIR filters for those applications requiring linear phase responses.

Cascade Form IIR Filtering

Filters implemented using the structure defined by equation (42-2) directly are known as *direct form* IIR filters. Direct form implementations are often sensitive to errors introduced by coefficient quantization and by computational, precision limits. Additionally, a filter designed to be stable can become unstable with increasing coefficient length, which is proportional to filter order.

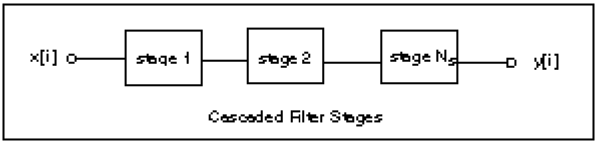
A less sensitive structure can be obtained by breaking up the direct form transfer function into lower order sections, or filter stages. The direct form transfer function of the filter given by equation (42-2) (with $a_0 = 1$) can be written as a ratio of z transforms, as follows:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_{N_b - 1} z^{-(N_b - 1)}}{1 + a_1 z^{-1} + \ldots + a_{N_a - 1} z^{-(N_a - 1)}} \quad . \qquad (42\text{-}2)$$

By factoring equation (42-2) into second-order sections, the transfer function of the filter becomes a product of second-order filter functions

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}} \qquad (42\text{-}3)$$

where $N_s = \lfloor N_a/2 \rfloor$ is the largest integer $\leq N_a/2$, and $N_a \geq Nb$. (N_s is the *number of stages.*) This new filter structure can be described as a *cascade* of second-order filters.



Each individual stage is implemented using the *direct form II* filter structure because it requires a minimum number of arithmetic operations and a minimum number of delay elements (internal filter states). Each stage has one input, one output, and two past internal states ($s_k[i\text{-}1]$ and $s_k[i\text{-}2]$).

If n is the number of samples in the input sequence, the filtering operation proceeds as in the following equations:

$y_0[i] = x[i]$,

$s_k[i] = y_{k-1}[i-1] - a_{1k} s_k[i-1] - a_{2k} s_k[i\text{-}2], \;\; k = 1, 2,..., N_s$

$y_k[i] = b_{Ok} s_k[i] + b_{1k} s_k[i\text{-}1] + b_{2k} s_k[i\text{-}2], \; k = 1, 2,..., N_s$

$y[i] = y_{Ns}[i]$

for each sample $i = 0, 1, 2,...,n\text{-}1$.

For filters with a single cutoff frequency (lowpass and highpass), second-order filter stages can be designed directly. The overall IIR lowpass or highpass filter contains cascaded second-order filters.

For filters with two cutoff frequencies (bandpass and bandstop), fourth-order filter stages are a more natural form. The overall IIR bandpass or bandstop filter is cascaded fourth-order filters. The filtering operation for fourth-order stages proceeds as in the following equations:

$y_0[i] = x[i],$

$s_k[i] = y_{k-1}[i-1] - a_{1k}s_k[i-1] - a_{2k}s_k[i\text{-}2] - a_{3k}s_k[i-3] - a_{4k}s_k[i-4],$
$\quad k = 1, 2,..., N_s$

$y_k[i] = b_{0k}s_k[i] + b_{1k}s_k[i\text{-}1] + b_{2k}s_k[i\text{-}2] + b_{3k}s_k[i\text{-}3] + b_{4k}s_k[i\text{-}4],$
$\quad k = 1, 2,..., N_s$

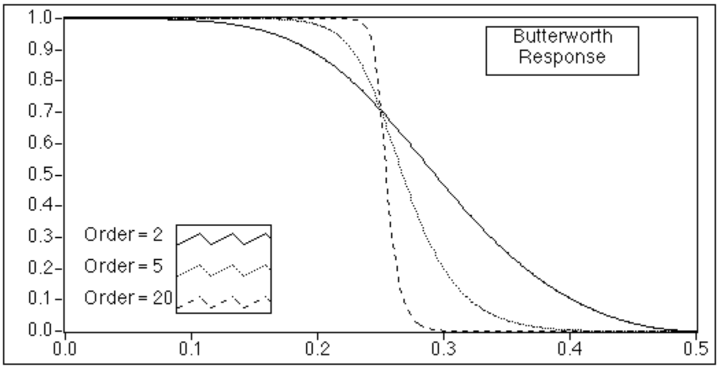
$y[i] = y_{Ns}[i].$

Notice that in the case of fourth-order filter stages, $N_s = \lfloor (N_a + 1)/4 \rfloor$.

Butterworth Filters

A smooth response at all frequencies a nd a monotonic decrease from the specified cutoff frequencies characterize the frequency response of Butterworth filters. Butterworth filters are maximally flat—the ideal response of unity in the passband and zero in the stopband. The half power frequency or the 3-dB down frequency corresponds to the specified cutoff frequencies.

The following illustration shows the response of a lowpass Butterworth filter. The advantage of Butterworth filters is a smooth, monotonically decreasing frequency response. After you set the cutoff frequency, LabVIEW sets the *steepness* of the transition proportional to the filter order. Higher order Butterworth filters approach the ideal lowpass filter response.

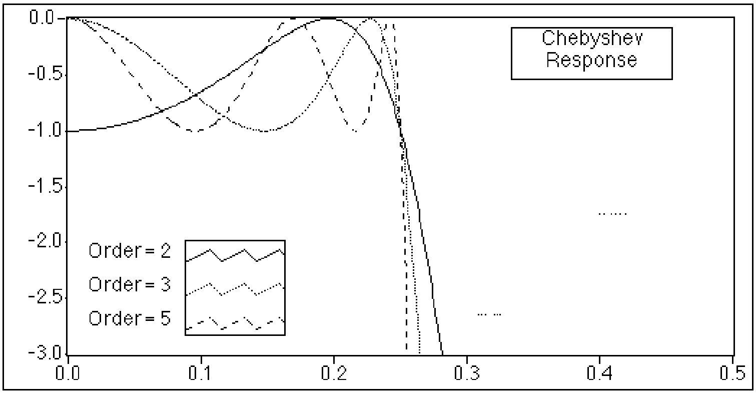


Chebyshev Filters

Butterworth filters do not always provide a good approximation of the ideal filter response because of the slow rolloff between the passband (the portion of interest in the spectrum) and the stopband (the unwanted portion of the spectrum).

Chebyshev filters minimize peak error in the passband by accounting for the maximum absolute value of the difference between the ideal filter and the filter response you want (the maximum tolerable error in the passband). The frequency response characteristics of Chebyshev filters have an equiripple magnitude response in the passband, monotonically decreasing magnitude response in the stopband, and a sharper rolloff than Butterworth filters.

The following graph shows the response of a lowpass Chebyshev filter. Notice that the equiripple response in the passband is constrained by the maximum tolerable ripple error and that the sharp rolloff appears in the stopband. The advantage of Chebyshev filters over Butterworth filters is that Chebyshev filters have a sharper transition between the passband and the stopband with a lower order filter. This produces smaller absolute errors and higher execution speeds.

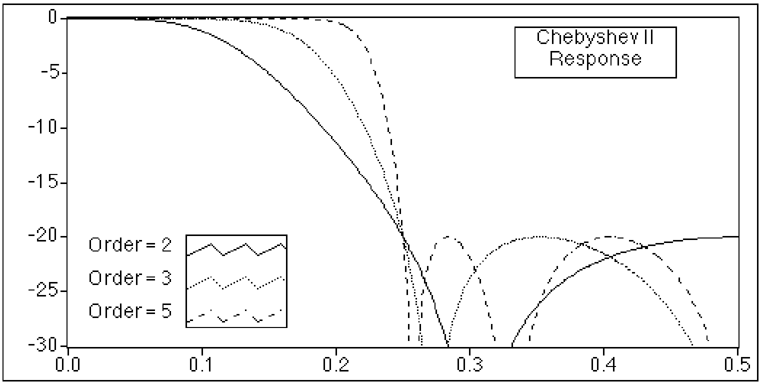


Chebyshev II or Inverse Chebyshev Filters

Chebyshev II, also known as inverse Chebyshev or Type II Chebyshev filters, are similar to Chebyshev filters, except that Chebyshev II filters distribute the error over the stopband (as opposed to the passband), and Chebyshev II filters are maximally flat in the passband (as opposed to the stopband).

Chebyshev II filters minimize peak error in the stopband by accounting for the maximum absolute value of the difference between the ideal filter and the filter response you want. The frequency response characteristics of Chebyshev II filters are equiripple magnitude response in the stopband, monotonically decreasing magnitude response in the passband, and a rolloff sharper than Butterworth filters.

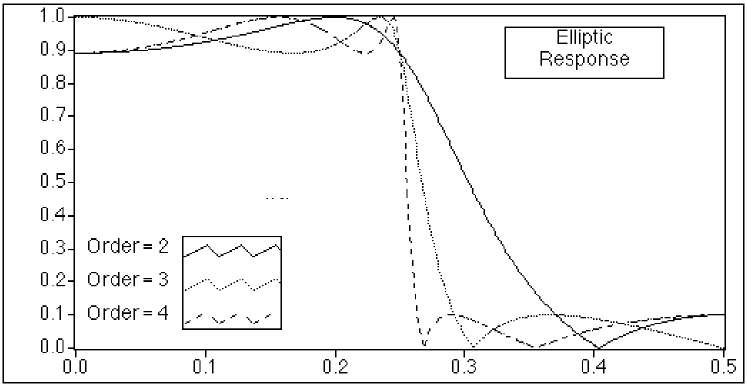
The following graph plots the response of a lowpass Chebyshev II filter. Notice that the equiripple response in the stopband is constrained by the maximum tolerable error and that the smooth monotonic rolloff appears in the stopband. The advantage of Chebyshev II filters over Butterworth filters is that Chebyshev II filters give a sharper transition between the passband and the stopband with a lower order filter. This difference corresponds to a smaller, absolute error and higher execution speed. One advantage of Chebyshev II filters over regular Chebyshev filters is that Chebyshev II filters distribute the error in the stopband instead of the passband.



Elliptic (or Cauer) Filters

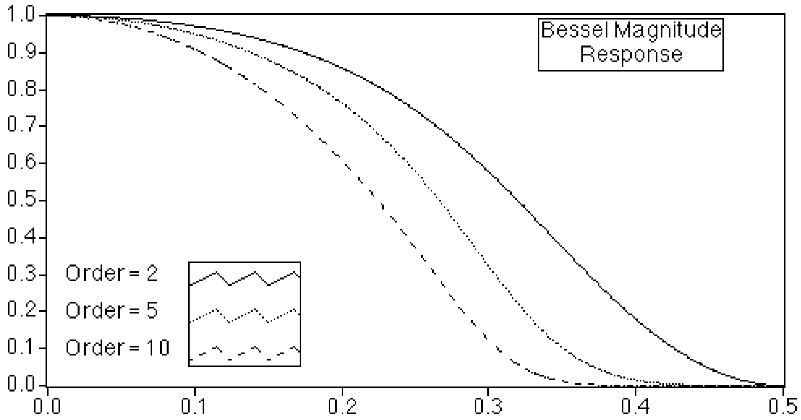
Elliptic filters minimize the peak error by distributing it over the passband and the stopband. Equi-ripples in the passband and the stopband characterize the magnitude response of elliptic filters. Compared with the same order Butterworth or Chebyshev filters, the elliptic design provides the sharpest transition between the passband and the stopband. For this reason, elliptic filters are used widely.

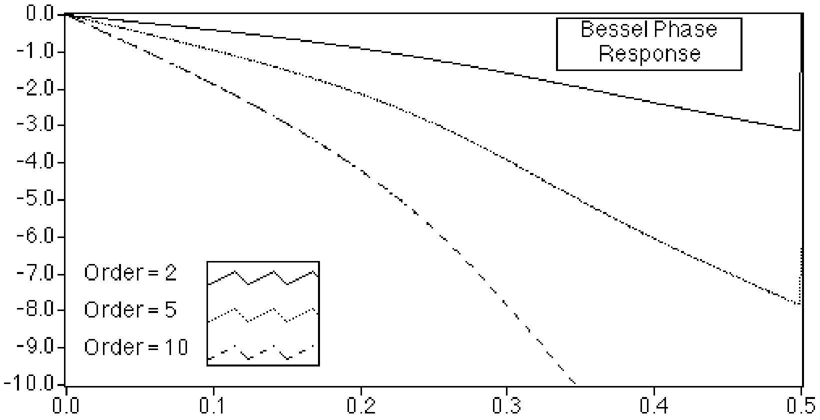
The following graph plots the response of a lowpass elliptic filter. Notice that the ripple in both the passband and stopband is constrained by the same maximum tolerable error (as specified by ripple amount in dB). Also, notice the sharp transition edge for even low-order elliptic filters.



Bessel Filters

You can use Bessel filters to reduce nonlinear phase distortion inherent in all IIR filters. In higher order filters and those with a steeper rolloff, this condition is more pronounced, especially in the transition regions of the filters. Bessel filters have maximally flat response in both magnitude and phase. Furthermore, the phase response in the passband of Bessel filters, which is the region of interest, is nearly linear. Like Butterworth filters, Bessel filters require high-order filters to minimize the error and, for this reason, are not widely used. You can also obtain linear phase response using FIR filter designs.The following graphs plot the response of a lowpass Bessel filter. Notice that the response is smooth at all frequencies, as well as monotonically decreasing in both magnitude and phase. Also, notice that the phase in the passband is nearly linear.





Finite Impulse Response Filters

Finite impulse response (FIR) filters are digital filters, which have a finite impulse response. FIR filters are also known as nonrecursive filters, convolution filters, or moving-average (MA) filters because you can express the output of an FIR filter as a finite convolution

$$y_i = \sum_{k=0}^{n-1} h_k x_{i-k}$$

where x represents the input sequence to be filtered, y represents the output filtered sequence, and h represents the FIR filter coefficients.

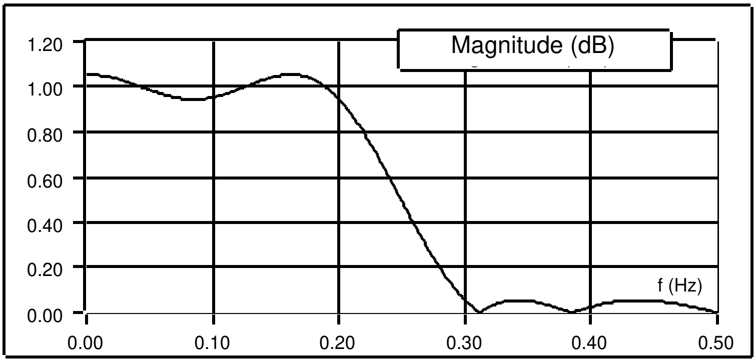
The following list gives the most important characteristics of FIR filters:

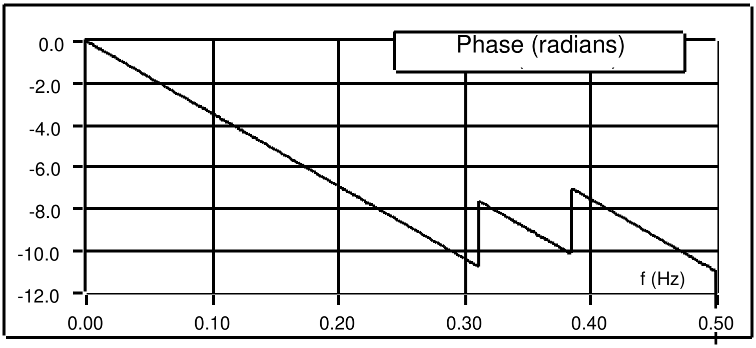
- They can achieve linear phase because of filter coefficient symmetry in the realization.
- They are always stable.
- You can perform the filtering function using the convolution and, as such, generally associate a delay with the output sequence

  $$\text{delay} = \frac{n-1}{2},$$

  where n is the number of FIR filter coefficients.

The following graphs plot a typical magnitude and phase response of FIR filters versus normalized frequency.





The discontinuities in the phase response arise from the discontinuities introduced when you compute the magnitude response using the absolute value. Notice that the discontinuities in phase are on the order of *pi*. The phase, however, is clearly linear. See Appendix D, *References*, for material that can give you more information on this topic.

You design FIR filters by approximating a specified, desired frequency response of a discrete-time system. The most common techniques approximate the desired magnitude response while maintaining a linear-phase response.

Designing FIR Filters by Windowing

The simplest method for designing linear-phase FIR filters is the *window design* method. To design a FIR filter by windowing, you start with an ideal frequency response, calculate its impulse response, and then truncate the impulse response to produce a finite number of coefficients. To meet the linear-phase constraint, by maintain symmetry about the center point of the coefficients. The truncation of the ideal impulse response results in the effect known as the Gibbs phenomenon – oscillatory behavior near abrupt transitions (cutoff frequencies) in the FIR filter frequency response.

You can reduce the effects of the Gibbs phenomenon by smoothing the truncation of the ideal impulse response using a smoothing window function. By tapering the FIR coefficients at each end, you can diminish the height of the side lobes in the frequency response. The disadvantage to this method, however, is that the main lobe widens, resulting in a wider transition region at the cutoff frequencies. The selection of a window function, then, is similar to the choice between Chebyshev and

Butterworth IIR filters in that it is a trade-off between side lobe levels near the cutoff frequencies and width of the transition region.

Designing FIR filters by windowing is simple and computationally inexpensive. It is therefore the fastest way to design FIR filters. It is not necessarily, however, the best FIR filter design method.

Designing Optimum FIR Filters using the Parks-McClellan Algorithm

The Parks-McClellan algorithm offers an optimum FIR filter design technique that attempts to design the best filter possible for a given number of coefficients. Such a design reduces the adverse effects at the cutoff frequencies. It also offers more control over the approximation errors in different frequency bands—control that is not possible with the window method.

Using the Parks-McClellan algorithm to design FIR filters is computationally expensive. This method, however, produces optimum FIR filters by applying time-consuming iterative techniques.

Designing Narrowband FIR Filters

When you use conventional techniques to design FIR filters with especially narrow bandwidths, the resulting filter lengths may be very long. FIR filters with long filter lengths often require lengthy design and implementation times, and are more susceptible to numerical inaccuracy. In some cases, conventional filter design techniques, such as the Parks-McClellan algorithm, may fail the design altogether.

You can use a very efficient algorithm, called the Interpolated Finite Impulse Response (IFIR) filter design technique, to design narrowband FIR filters. Using this technique produces narrowband filters that require far fewer coefficients (and therefore fewer computations) than those filters designed by the direct application of the Parks-McClellan algorithm. LabVIEW also uses this technique to produce wideband, lowpass (cutoff frequency near Nyquist) and highpass filters (cutoff frequency near zero). For more information about IFIR filter design, see *Multirate Systems and Filter Banks* by P.P. Vaidyanathan, or the paper on interpolated finite impulse response filters by Neuvo, et al., listed in Appendix D, *References*, of this manual.

Windowed FIR Filters

You use the **filter type** parameter of the FIR VIs to select the type of windowed FIR filter you want: lowpass, highpass, bandpass, or bandstop. The following list gives the two related FIR VIs:

- FIR Windowed Coefficients—Generates the windowed (or unwindowed) coefficients.
- FIR Windowed Filters—Filters the input using windowed (or unwindowed) coefficients.

Optimum FIR Filters

You can use the Parks-McClellan algorithm to design optimum, linear-phase, FIR filter coefficients in the sense that the resulting filter optimally matches the filter specifications for a given number of coefficients. The Parks-McClellan VI takes as input an array of band descriptions, each containing information describing the response you want for the given band. The VI outputs the FIR coefficients along with computed ripple, which is a measure of the deviation of the resulting filter from the ideal filter specifications.

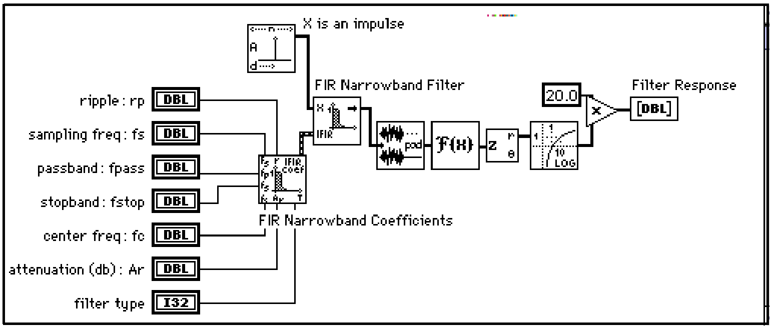
Four VIs use the Parks-McClellan VI to implement filters whose stopband and passband ripple level are equal: Equiripple LowPass, Equiripple HighPass, Equiripple BandPass, and Equiripple BandStop.

FIR Narrowband Filters

You can design narrowband FIR filters using the FIR Narrowband Coefficients VI, and then implement the filtering using the FIR Narrowband Filter VI. The design and implementation are separate operations because many narrowband filters require lengthy design times, while the actual filtering process is very fast and efficient. Keep this in mind when creating your narrowband filtering diagrams.

The parameters required for narrowband filter specification are filter type, sampling rate, passband and stopband frequencies, passband ripple (linear scale), and stopband attenuation (decibels). For bandpass and bandstop filters, passband and stopband frequencies refer to bandwidths, and you must specify an additional center frequency parameter. You can also design wideband lowpass filters (cutoff frequency near Nyquist) and wideband highpass filters (cutoff frequency near zero) using the narrowband filter VIs.

The following illustration shows how to use the FIR Narrowband Coefficients VI and the FIR Narrowband Filter VI to estimate the response of a narrowband filter to an impulse.



Nonlinear Filters

Smoothing windows, IIR filters, and FIR filters are linear because they satisfy the superposition and proportionality principles

$$\mathrm{L}\ \{\mathrm{a}x(t) + \mathrm{b}y(t)\} = \mathrm{aL}\ \{x(t)\} + \mathrm{bL}\{y(t)\},$$

where a and b are constants, $x(t)$ and $y(t)$ are signals, L{•} is a linear filtering operation, and their inputs and outputs are related via the convolution operation.

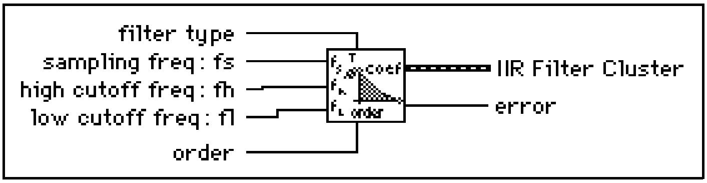
A nonlinear filter does not meet the preceding conditions and you cannot obtain its output signals via the convolution operation, because a set of coefficients cannot characterize the impulse response of the filter. Nonlinear filters provide specific filtering characteristics that are difficult to obtain using linear techniques. The median filter is a nonlinear filter that combines lowpass filter characteristics (to remove high-frequency noise) and high-frequency characteristics (to detect edges).

Filter VI Descriptions

The following Filter VIs are available.

Bessel Coefficients

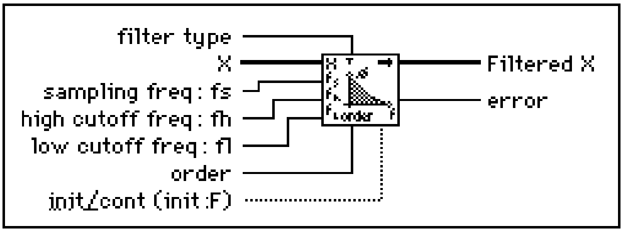
Generates the set of filter coefficients to implement an IIR filter as specified by the Bessel filter model. You can then pass these coefficients to the IIR Filter VI.



The Bessel Coefficients VI is a subVI of the Bessel Filter VI.

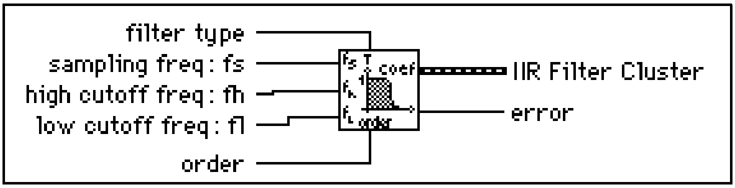
Bessel Filter

Generates a digital, Bessel filter using the filter type, sampling frequency, high cutoff frequency, low cutoff frequency, and order by calling the Bessel Coefficients VI. The VI then calls the IIR filter to filter the **X** sequence using this model to obtain a Bessel **Filtered X** sequence.



Butterworth Coefficients

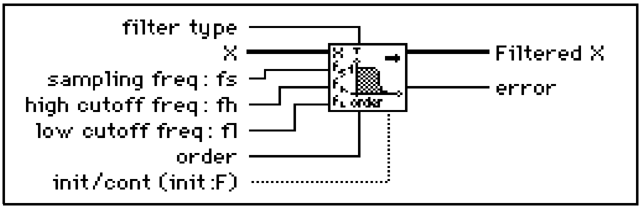
Generates the set of filter coefficients to implement an IIR filter as specified by the Butterworth filter model. You can pass these filter coefficients (**IIR Filter Cluster**) to the IIR Cascade Filter VI to filter a sequence of data.



This VI is a subVI of the Butterworth Filter VI.

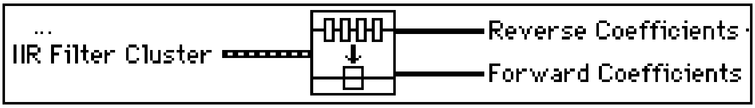
Butterworth Filter

Generates a digital Butterworth filter using the sampling frequency, low cutoff frequency, high cutoff frequency, order, and filter type by calling the Butterworth Coefficients VI. The Butterworth Filter VI then calls the IIR Filter VI to filter the **X** sequence using this model to get a Butterworth **Filtered X** sequence.



Cascade—>Direct Coefficients

Converts IIR filter coefficients from the cascade form to the direct form.



As an example, you can convert a cascade filter, composed of two second-order stages, to a direct form filter as follows:

Reverse Coefficients:

$\{a_{11},a_{21},a_{12},a_{22}\}$ ->$\{1.0,a_1,a_2,a_3,a_4\}$

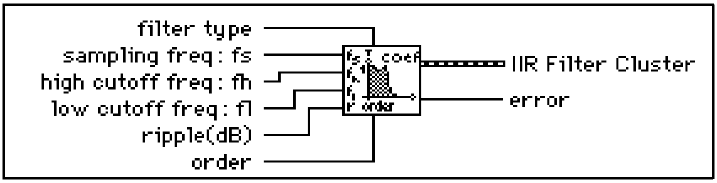
Forward Coefficients:

$\{b_{01},b_{11},b_{21},b_{02},b_{12},b_{22}\}$ ->$\{b_0,b_1,b_2,b_3,b_4\}$

See the IIR Cascade Filter VI for information about cascade form filtering, the IIR Filter VI for information on direct form filtering, and the *About Digital Filtering Functions* section of this chapter for a discussion of both filter forms.

Chebyshev Coefficients

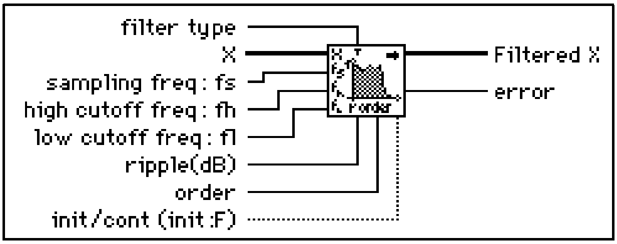
Generates the set of filter coefficients to implement an IIR filter as specified by the Chebyshev filter model. You can pass these coefficients to the IIR Filter VI to filter a sequence of data.



The Chebyshev Coefficients VI is a subVI of the Chebyshev Filter VI.

Chebyshev Filter

Generates a digital, Chebyshev filter using the sampling frequency, lower cutoff frequency, upper cutoff frequency, ripple, order, and filter type by calling the Chebyshev Coefficients VI. The Chebyshev Filter VI filters the **X** sequence using this model to obtain a Chebyshev **Filtered X** sequence by calling the IIR Filter VI.

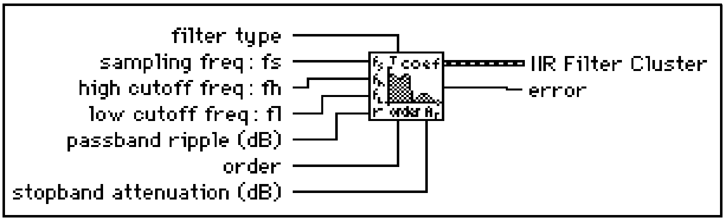


Convolution

For information on Convolution, see Chapter 40, *Analysis Digital Signal Processing VIs*, in this manual.

Elliptic Coefficients

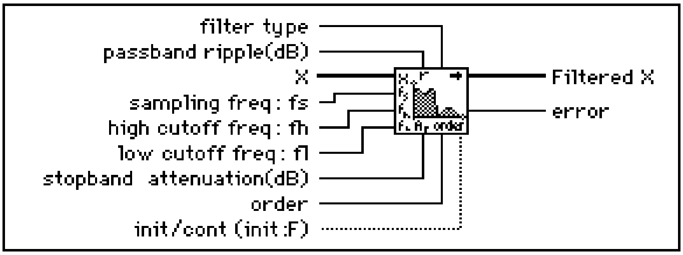
Generates the set of filter coefficients to implement a digital elliptic IIR filter. You can pass these coefficients to the IIR Filter VI.



The Elliptic Coefficients VI is a subVI of the Elliptic Filter VI.

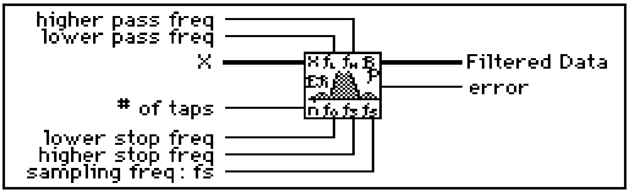
Elliptic Filter

Generates a digital, elliptic filter using the **sampling frequency, lower cutoff frequency, upper cutoff frequency, filter type, passband ripple, stopband attenuation**, and **order** by calling the Elliptic Coefficients VI. The Elliptic Filter VI then calls the IIR Filter VI to filter the **X** sequence using this model to obtain an elliptic **Filtered X** sequence.



Equiripple BandPass

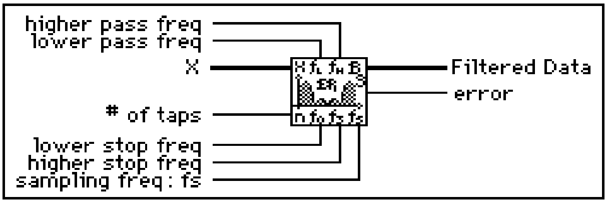
Generates a bandpass FIR filter with equi-ripple characteristics using the Parks-McClellan algorithm and the **higher pass frequency, lower pass frequency, # of taps, lower stop frequency, higher stop frequency**, and **sampling frequency**. The VI then filters the input sequence **X** to obtain the bandpass, filtered, linear-phase sequence **Filtered Data**.



The first stopband of the filter region goes from zero (DC) to the lower stop frequency. The passband region goes from the lower pass frequency to the higher pass frequency, and the second stopband region goes from the higher stop frequency to the Nyquist frequency.

Equiripple BandStop

Generates a bandstop FIR digital filter with equi-ripple characteristics using the Parks-McClellan **algorithm** and **higher pass frequency**, **lower pass frequency**, **# of taps**, **lower stop frequency**, **higher stop frequency**, and **sampling frequency**. The VI then filters the input sequence X to obtain the bandstop, filtered, linear-phase sequence Filtered Data.

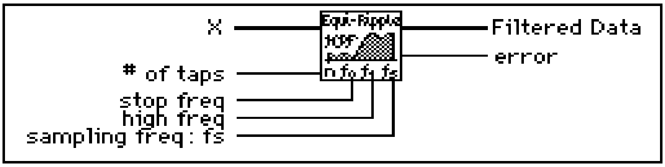


The first passband region of the filter goes from zero (DC) to the lower pass frequency. The stopband region goes from the lower stop frequency to the higher stop frequency, and the second passband region goes from the higher pass frequency to the Nyquist frequency.

Equiripple HighPass

Generates a highpass FIR filter with equi-ripple characteristics using the Parks-McClellan algorithm and the **# of taps, stop frequency, high frequency**, and

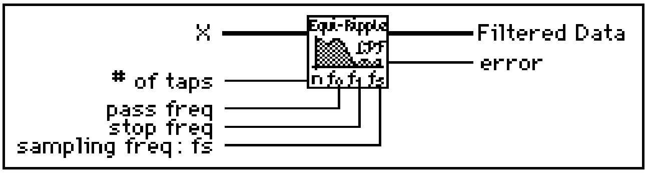
sampling frequency. The VI then filters the input sequence X to obtain the highpass, filtered, linear-phase sequence **Filtered Data**.



The stopband of the filter goes from zero (DC) to the stop frequency. The transition band goes from the stop frequency to the high frequency, and the passband goes from the high frequency to the Nyquist frequency.

Equiripple LowPass

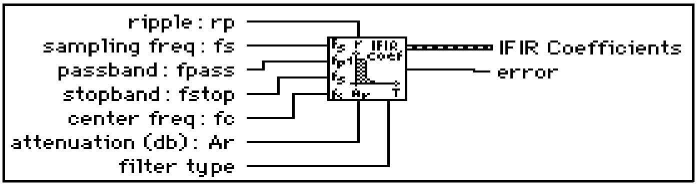
Generates a lowpass FIR filter with equiripple characteristics using the Parks-McClellan algorithm and the **# of taps**, **pass frequency**, **stop frequency**, and **sampling frequency**. The VI then filters the input sequence **X** to obtain the lowpass filtered, linear-phase sequence **Filtered Data**.



The passband of the filter goes from zero (DC) to **pass freq**. The transition band goes from **pass freq** to **stop freq**, and the stopband goes from **stop freq** to the Nyquist frequency.

FIR Narrowband Coefficients

Generates a set of filter coefficients to implement a digital interpolated FIR filter. You can pass these coefficients to the FIR Narrowband Filter VI to filter the data.



The following figures show how the narrowband filter parameters define the lowpass, highpass, bandpass, and bandstop filters. The filter response on the y axis is shown on a linear scale. For this reason, the stopband attenuation A_r was mapped to a linear attenuation using the following equations:

$$A_r = -20\log\langle\delta_A\rangle$$

$$\delta_A = 10\frac{-Ar}{20}.$$

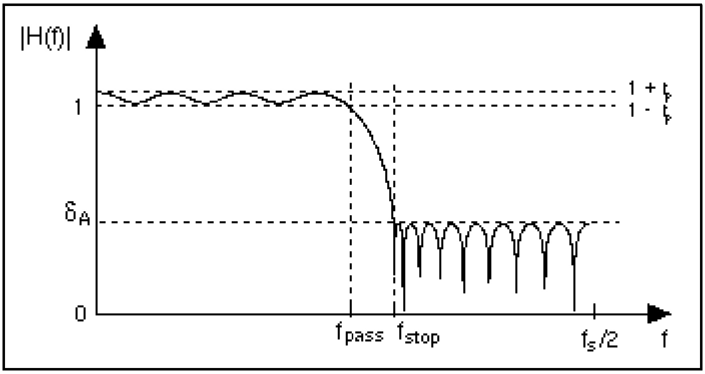


Figure 42-1. Lowpass Filter

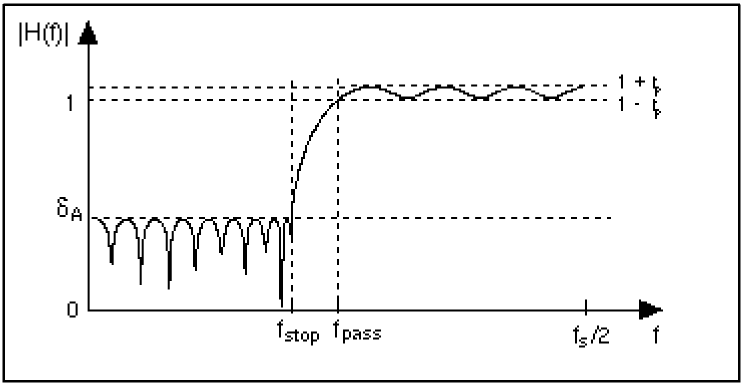


Figure 42-2. Highpass Filter

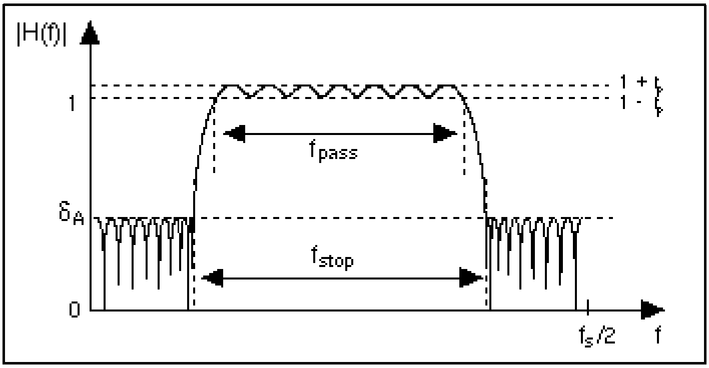


Figure 42-3. Bandpass Filter

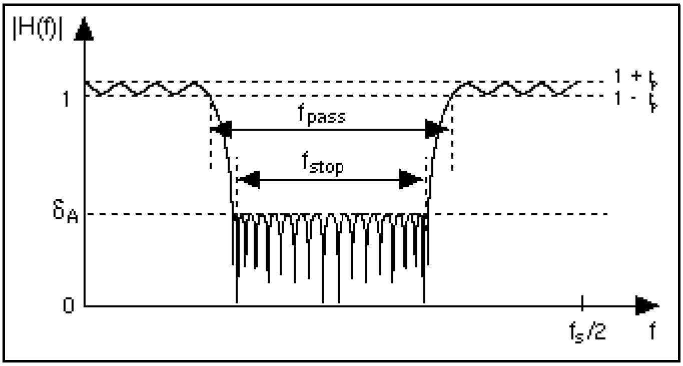
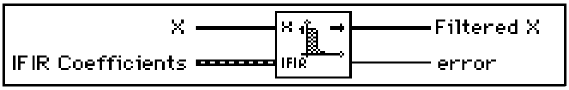


Figure 42-4. Bandstop Filter

FIR Narrowband Filter

Filters the input sequence **X** using the IFIR filter specified by **IFIR Coefficients** as designed by the FIR Narrowband Filter Coefficients VI.



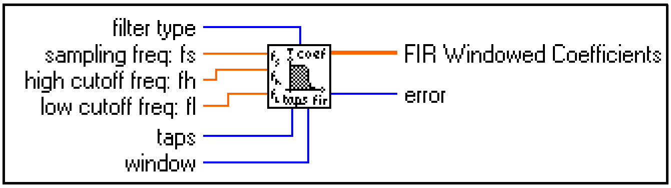
Note: ***The overall filter is a linear-phase FIR filter. The delay for this filter is***

$$\frac{[(N_G - 1) \bullet M + N_I]}{2}$$

where N_G is the number of elements in the array Model Filter, N_I is the number of elements in the array Image Suppressor, and M is the value of interpolation in the cluster IFIR Coefficients.

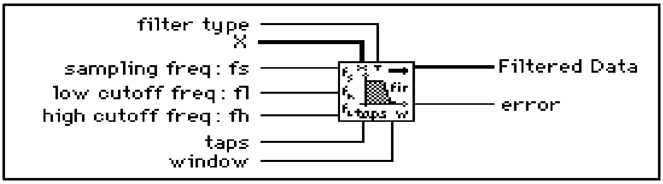
FIR Windowed Coefficients

Generates the set of filter coefficients you need to implement a FIR windowed filter.



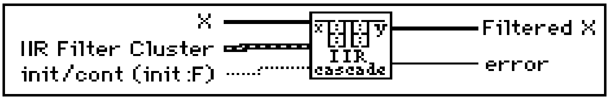
FIR Windowed Filter

Filters the input data sequence, **X**, using the set of windowed FIR filter coefficients specified by the **sampling frequency**, **cutoff frequency**, and number of **taps**.

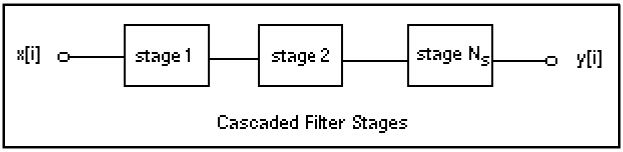


IIR Cascade Filter

Filters the input sequence **X** using the cascade form of the IIR filter specified by the **IIR Filter Cluster**.



This IIR implementation is called *cascade* because it is a cascade of second- or fourth-order filter stages. The output of one filter stage is the input to the next filter stage for all N_s filter stages.



Second-Order Filtering

Each second-order stage (stage number $k = 1,2,...N_s$) has two reverse coefficients (a_{1k},a_{2k}), and three forward coefficients (b_{0k},b_{1k},b_{2k}). The total number of reverse coefficients is $2Ns$ and the total number of forward coefficients is $3Ns$. The **Reverse Coefficients** and the **Forward Coefficients** array contain the coefficients for one stage followed by the coefficients for the next stage, and so on. For example, an IIR filter composed of two second-order stages must have a total of four reverse coefficients and six forward coefficients, as follows:

Reverse Coefficients = $\{a_{11}, a_{21}, a_{12}, a_{22}\}$

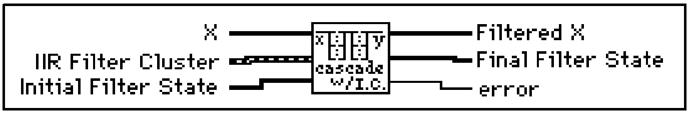
Forward Coefficients = $\{b_{01}, b_{11}, b_{21}, b_{02}, b_{12}, b_{22}\}$

Fourth-Order Filtering

For fourth order cascade stages, the filtering is implemented in the same manner as in the second-order stages, but each stage must have four reverse coefficients ($a_1k...a_4k$) and five forward coefficients ($b_{0k}...b_{4k}$).

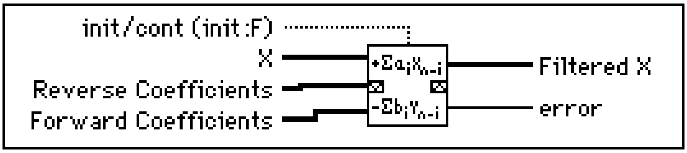
IIR Cascade Filter with Integrated Circuit

Filters the input sequence, **X**, using the cascade form of the IIR filter specified by the **IIR Filter Cluster**.



IIR Filter

Filters the input sequence **X** using the direct form IIR filter specified by **Reverse Coefficients** and **Forward Coefficients**.



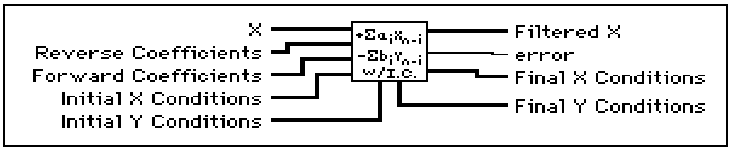
If y represents the output sequence **Filtered X**, the VI obtains the elements of y using

$$y_i = \frac{1}{a_0}\left(\sum_{j=0}^{n-1} b_j x_{i-j} - \sum_{k=1}^{m-1} a_k y_{i-k}\right),$$

where *n* is the number of **Forward Coefficients** (represented by b_j), and *m* is the number of **Reverse Coefficients** (represented by a_k).

IIR Filter with Integrated Circuit

Filters the input sequence **X** using the direct form IIR filter specified by **Reverse Coefficients** and **Forward Coefficients**.



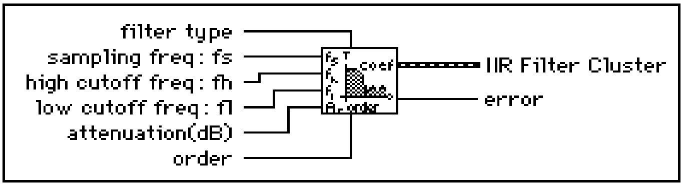
If *y* represents the output sequence **Filtered X**, the VI obtains the elements of *y* using

$$y_i = \frac{1}{a_0}\left(\sum_{j=0}^{n-1} b_j x_{i-j} - \sum_{k=1}^{m-1} a_k y_{i-k}\right),$$

where *n* is the number of **Forward Coefficients** (represented by *bj*), and *m* is the number of **Reverse Coefficients** (represented by a_k).

Inv Chebyshev Coefficients

Generates the set of filter coefficients to implement an IIR filter as specified by the Chebyshev II Filter model. You can pass these coefficients to the IIR Filter VI to filter a sequence of data.

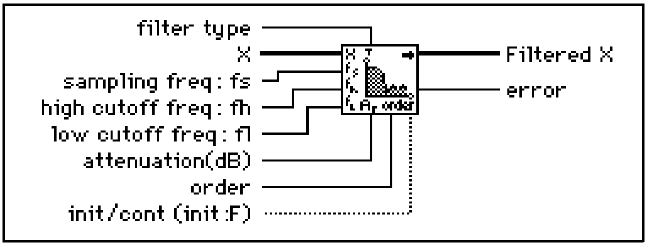


The Inv Chebyshev Coefficients VI is a subVI of the Inverse Chebyshev Filter VI.

Inverse Chebyshev Filter

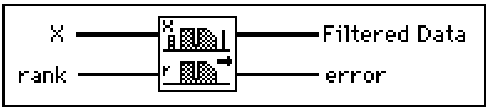
Generates a digital, Chebyshev II filter using the specified sampling frequency, cutoff frequencies, **attenuation** in decibels, **filter type**, and filter **order** by calling the Inv Chebyshev Coefficients VI. The Inverse Chebyshev Filter VI filters the input sequence

X using this model to obtain a Chebyshev II **Filtered X** sequence by calling the IIR Filter VI.



Median Filter

Applies a median filter of **rank** to the input sequence **X**.



If Y represents the output sequence **Filtered Data**, and if J_i represents a subset of the input sequence **X** centered about the i^{th} element of **X**

$J_i = \{x_{i-r}, x_{i-r+1}, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{i+r-1}, x_{i+r}\}$,

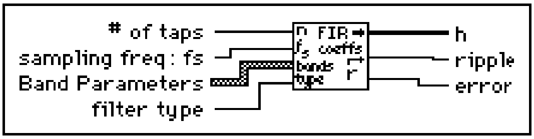
and if the indexed elements outside the range of **X** equal zero, the VI obtains the elements of y using

$y_i = \text{Median}(J_i) \quad \text{for } i = 0, 1, 2, \ldots, n-1,$

where n is the number of elements in the input sequence **X**, and r is the filter **rank**.

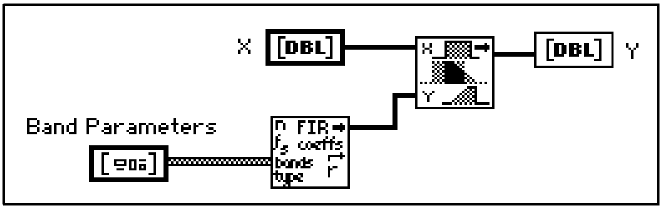
Parks-McClellan

Generates a set of linear-phase FIR multiband digital filter coefficients using the number of **taps**, **sampling frequency**, **Band Parameters**, and **filter type**.



Note: ***This VI finds the coefficients using iterative techniques based upon an error criterion. Although you specify valid filter parameters, the algorithm may fail to converge.***

This VI generates only the filter coefficients. It does not perform the filtering function. To filter a sequence **X** using the set of FIR filter coefficients **h**, use the Convolution VI with **X** and **h** as the input sequences.



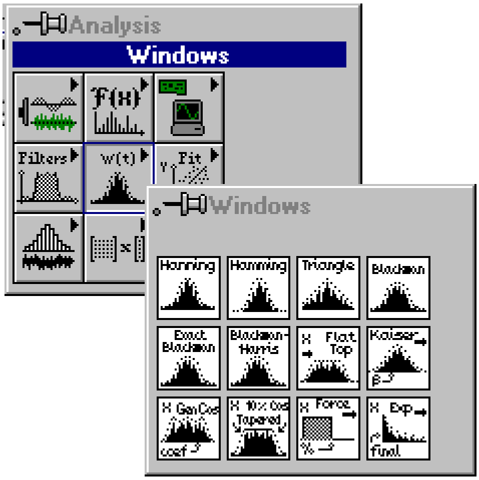
The equiripple filters use a similar technique to filter the data.

Chapter 43

Analysis Window VIs

This chapter describes the VIs that implement smoothing windows.

To access the Window palette, select **Function»Analysis»Windows**. The following illustration shows the options that are available on the Windows palette.



For examples of how to use the window VIs, see the examples located in `examples\analysis\windxmpl.llb`.

Introduction to Smoothing Windows

In practical, signal-sampling applications, you can obtain only a finite record of the signal, even when you carefully observe the sampling theorem and sampling conditions. Unfortunately for the discrete-time system, the finite sampling record results in a truncated waveform that has different spectral characteristics from the original continuous-time signal. These discontinuities produce leakage of spectral information, resulting in a discrete-time spectrum that is a smeared version of the original continuous-time spectrum.

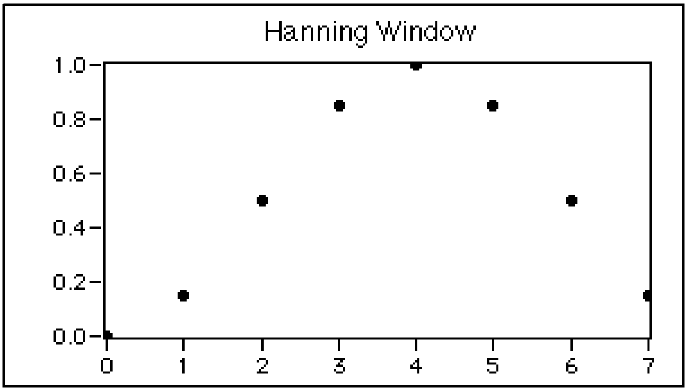
A simple way to improve the spectral characteristics of a sampled signal is to apply smoothing windows. When performing Fourier or spectral analysis on finite-length data, you can use windows to minimize the transition edges of your truncated waveforms, thus reducing spectral leakage. When used in this manner, smoothing windows act like predefined, narrowband, lowpass filters.

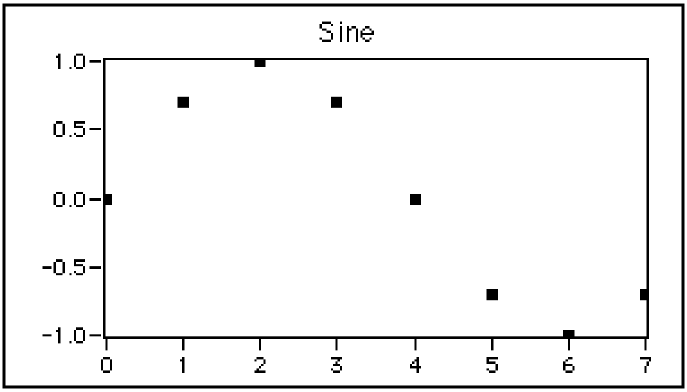
Windows for Spectral Analysis versus Windows for Coefficient Design

The window VIs implemented in the Analysis library in LabVIEW are designed for spectral analysis applications. In these applications, the input signal is windowed by passing it through one of the window VIs. The windowed signal is then passed to a DFT-based VI for frequency-domain display and analysis.

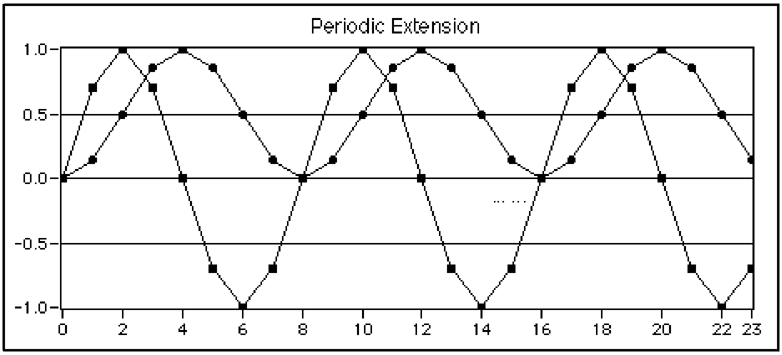
The window functions designed for spectral analysis must be *DFT-even*, a term defined by Fredric J. Harris in his paper *On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform* (*Proceedings of the IEEE*, Volume 66, No.1, January 1978). A window function is DFT-even if its dot product (inner product) with integral cycles of sine sequences is identically zero. Another way to think of a DFT-even sequence is that its DFT has no imaginary component.

The following figures illustrate the Hanning window and one cycle of a sine pattern for a sample size of 8. The figures below show that the DFT-even Hanning window is not symmetric about its midpoint and its last point is not equal to its first point, much like one complete cycle of a sine pattern.





Finally, the DFT considers input sequences to be periodic—that the signal being analyzed is actually a concatenation of the input signal. The following illustration shows three such cycles of the previous sequences, demonstrating the smooth periodic extension of the DFT-even window and the single-cycle sine pattern.



Another type of window application is that of FIR filter design (see the descriptions of *FIR Windowed Coefficients* and *FIR Windowed Filter*). This application requires windows that are symmetric about their midpoint.

The following equations of the Hanning window function illustrate the difference between the DFT-even window function (spectral analysis) and the symmetrical window function (coefficient design).

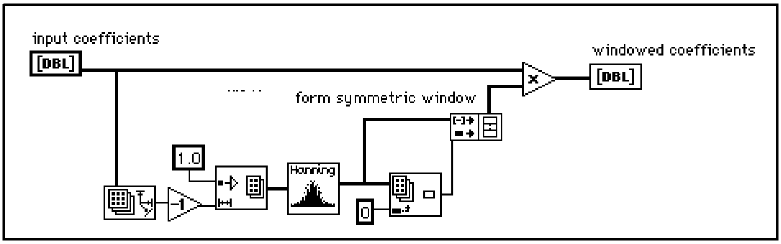
Hanning window function for spectral analysis:

$$w[i] = 0.5\left(1 - \cos\left(\frac{2\pi i}{N}\right)\right) \text{ for } i\text{=0,1, 2, ..., } N\text{–1}$$

Hanning window function for symmetrical coefficient design:

$$w[i] = 0.5\left(1 - \cos\left(\frac{2\pi i}{N-1}\right)\right) \text{ for } i\text{=0, 1, 2, ..., } N\text{–1}$$

The two equations above show that you can implement the symmetrical window functions by slightly modifying the use of the DFT-even window functions. The following illustration shows a block diagram that uses the Hanning Window VI to implement symmetrical windowing of filter coefficients.



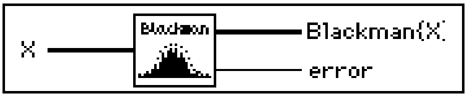
See Appendix D, *References*, for more information on smoothing windows.

Window VI Descriptions

The following Window VIs are available.

Blackman Window

Applies a Blackman window to the input sequence **X**.



If y represents the output sequence **Blackman{X}**, the VI obtains the elements of y from

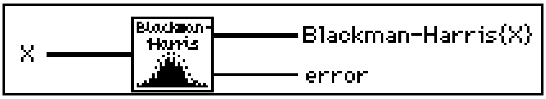
$$y_i = x_i\,[0.42 - 0.50\cos(w) + 0.08\cos(2w)] \quad \text{for } i = 0, 1, 2, \ldots, n-1,$$

$$w = \frac{2\pi i}{n} \ ,$$

where n is the number of elements in **X**.

Blackman-Harris Window

Applies a three-term, Blackman-Harris window to the input sequence **X**.



If Y represents the output sequence **Blackman-Harris{X}**, the VI obtains the elements of Y from

$$y_i = x_i\,[0.42323 - 0.49755\cos(w) + 0.07922\cos(2w)]$$

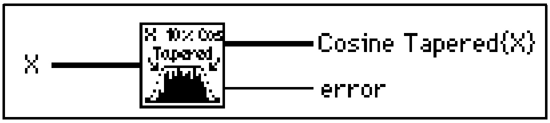
for $i = 0, 1, 2, \ldots, n-1$

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in **X**.

Cosine Tapered Window

Applies a cosine tapered window to the input sequence **X**.



If Y represents the output sequence **Cosine Tapered{X}**, the VI obtains the elements of Y from

$$y_i = \begin{cases} 0.5x_i(1 - \cos w) & \text{for i = 0, 1, 2,..., m-1, and for i = n-m, n-m+1,..., n-1} \\ x_i & \text{elsewhere} \end{cases}$$

where $w = \dfrac{2\pi i}{n}$,

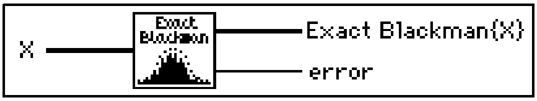
$m = \text{round}\left(\dfrac{n}{10}\right)$, and

where n is the number of elements in the input sequence **X**.

Using this window is the equivalent of applying the Hanning window to the first and last 10% of the input sequence **X**.

Exact Blackman Window

Applies an Exact Blackman window to the input sequence **X**.



If *Y* represents the output sequence **Exact Blackman{X}**, the VI obtains the elements of *Y* from

$y_i = x_i\,[a_0 - a_1 \cos(w) + a_2 \cos(2w)]$

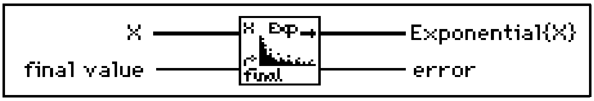
for $i = 0, 1, 2, \ldots, n{-}1$

$$w = \frac{2\pi i}{n},$$

where *n* is the number of elements in **X**, $a_0 = 7938/18608$, $a_1 = 9240/18608$, and $a_2 = 1430/18608$.

Exponential Window

Applies an exponential window to the input sequence **X**.



If *y* represents the output sequence **Exponential{X}**, the VI obtains the elements of *y* from

$y_i = x_i \exp(ai)$ for $i = 0, 1, 2, \ldots, n{-}1$,

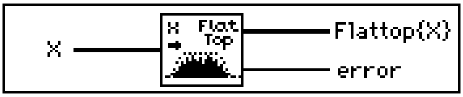
$$a = \frac{ln(f)}{n-1},$$

where *f* is the **final value**, and *n* is the number of samples in **X**.

You can use this VI to analyze transients.

Flat Top Window

Applies a flat top window to the input sequence **X**.



If *Y* represents the output sequence **Flattop{X}**, the VI obtains the elements of *Y* from

$y_i = x_i\,[0.2810639 - 0.5208972 \cos(w) + 0.1980399 \cos(2w)]$

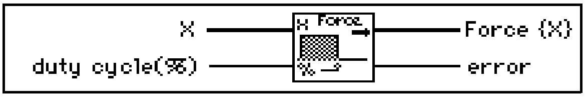
for $i = 0, 1, 2, \ldots, n-1$

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in **X**.

Force Window

Applies a force window to the input sequence **X**.



If Y represents the output sequence **Force{X}**, the VI obtains the elements of Y from

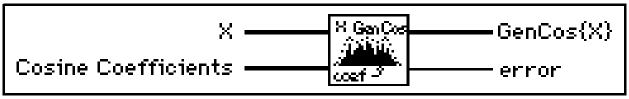
$$y_i = \begin{cases} x_i & (\text{if } 0 \le i \le d) \\ 0 & \text{elsewhere} \end{cases} \qquad \text{for i = 0, 1, 2, ..., n-1}$$

$d = (0.01)(n)($**duty cycle**$)$, where n is the number of elements in **X**.

You also can use this VI to analyze transients.

General Cosine Window

Applies a general, cosine window to the input sequence **X**.



If a represents the **Cosine Coefficients** input sequence and y represents the output sequence **GenCos{X}**, the VI obtains the elements of y from

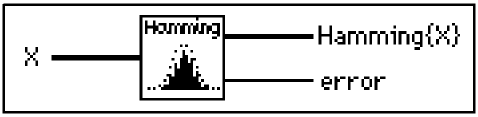
$$y_i = x_i \sum_{k=0}^{m-1} (-1)^k a_k \cos(kw) \qquad \text{for } i = 0, 1, 2, \ldots, n-1$$

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in **X**, and m is the number of **Cosine Coefficients**.

Hamming Window

Applies a Hamming window to the input sequence **X**.



If y represents the output sequence **Hamming{X}**, the VI obtains the elements of y from

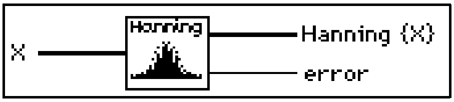
$y_i = x_i\,[0.54 - 0.46\cos(w)]$ for $i = 0, 1, 2, \ldots, n-1$,

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in the input sequence **X**.

Hanning Window

Applies a Hanning window to the input sequence **X**.



If Y represents the output sequence **Hanning {X}**, the VI obtains the elements of Y using

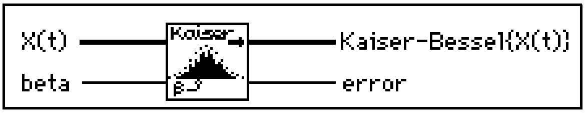
$y_i = 0.5\,x_i\,[1 - \cos(w)]$ for $i = 0, 1, 2, \ldots, n-1$,

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in **X.**

Kaiser-Bessel Window

Applies a Kaiser-Bessel window to the input sequence **X(t)**.



If y represents the output sequence, **Kaiser-Bessel{X(t)}**, the VI obtains the elements of y from

$$y_i = x_i \frac{I_o(\beta\sqrt{1.0 - a^2})}{I_o(\beta)} \quad \text{for } i = 0, 1, 2, \ldots n - 1$$

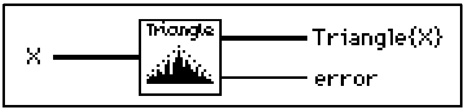
$$a = \frac{i-k}{k},$$

$$k = \frac{n-1}{2},$$

where n is the number of elements in **X(t)**, and $I_O(\bullet)$ is the zero-order modified Bessel function.

Triangle Window

Applies a triangular window to the input sequence **X**.



Note: ***The triangle smoothing window is also known as the Bartlett smoothing window.***

If y represents the output sequence **Triangle{X}**, the VI obtains the elements of y from

$y_i = x_i \operatorname{tri}(w)$ for $i = 0, 1, 2, \ldots, n–1$,

$$w = \frac{2i-n}{n},$$

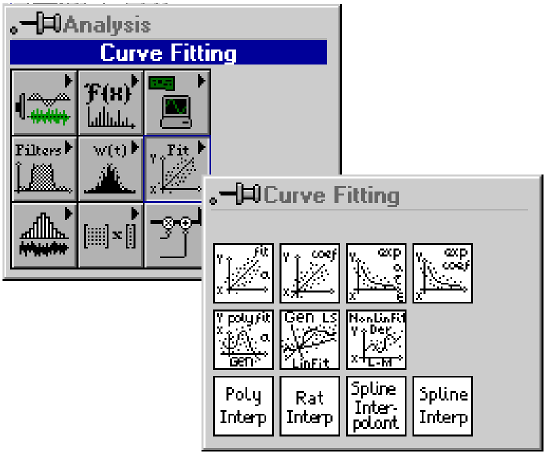
where $\operatorname{tri}(w) = 1 - |w|$, and n is the number of elements in **X**.

Chapter 44

Analysis Curve-Fitting VIs

This chapter describes the VIs that perform curve fitting analysis or regression.

To access the **Curve-Fitting** palette, choose **Functions»Analysis»Curve Fitting**, as shown in the following illustration.



For examples of how to use the regression VIs, see the examples located in `examples\analysis\regressn.llb`.

Introduction to Curve Fitting

Curve fitting analysis is a technique for extracting a set of curve parameters or coefficients from the data set to obtain a functional description of the data set. The algorithm that fits a curve to a particular data set is known as the Least Squares Method and is discussed in most introductory textbooks in probability and statistics. The error is defined as

$$e(a) = [f(x,a) - y(x)]^2, \tag{44-1}$$

where $e(a)$ is the error, $y(x)$ is the observed data set, $f(x,a)$ is the functional description of the data set, and a is the set of curve coefficients which best describes the curve.

For example, let $a = \{a_0, a_1\}$. Then the functional description of a line is

$$f(x,a) \quad = a_0 + a_1 x.$$

The least squares algorithm finds a by solving the system

$$\frac{\partial}{\partial a} e(a) = 0 \qquad (44\text{-}2)$$

To solve this system, you set up and solve the Jacobian system generated by expanding equation (44-2). After you solve the system for a, you can obtain an estimate of the observed data set for any value of x using the functional description $f(x, a)$.

In LabVIEW, the curve fitting VIs automatically set up and solve the Jacobian system and return the set of coefficients that best describes your data set. You can concentrate on the functional description of your data and not worry about solving the system in equation (44-2).

Two input sequences, Y Values and X Values, represent the data set $y(x)$. A sample or point in the data set is

$$(x_i, y_i),$$

where x_i is the i^{th} element of the sequence X Values, and y_i is the i^{th} element of the sequence Y Values.

In general, for each predefined type of curve fit, there are two types of VIs, unless otherwise specified. One type returns only the coefficients, so that you can further manipulate the data. The other type returns the coefficients, the corresponding expected or fitted curve, and the mean squared error (MSE). Because it is a discrete system, the VI calculates the MSE, which is a relative measure of the residuals between the expected curve values and the actual observed values, using the formula

$$\text{MSE} = \frac{1}{n}\sum_{i=0}^{n-1}(f_i - y_i)^2 \qquad (44\text{-}3)$$

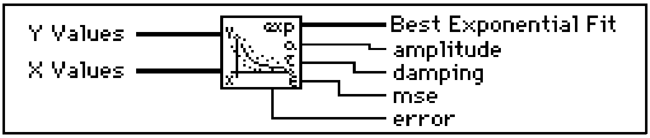
where *f* is the sequence representing the fitted values, *y* is the sequence representing the observed values, and *n* is the number of sample points observed.

Curve Fitting VI Descriptions

The following Curve Fitting VIs are available.

Exponential Fit

Finds the exponential curve values and the set of exponential coefficients **amplitude** and **damping**, which describe the exponential curve that best represents the input data set.



The general form of the exponential fit is given by

$F = ae^{\tau X}$,

where *F* is the output sequence **Best Exponential Fit**, *X* is the input sequence **X Values**, *a* is the **amplitude**, and τ is the **damping** constant.

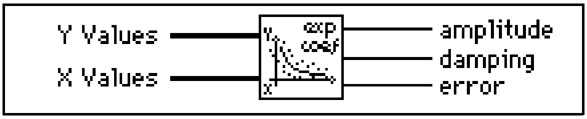
The VI obtains **mse** using the formula

$$\text{mse} = \frac{1}{n}\sum_{i=0}^{n-1}(f_i - y_i)^2 \quad ,$$

where *f* is the output sequence **Best Exponential Fit**, *y* is the input sequence **Y Values**, and *n* is the number of data points.

Exponential Fit Coefficients

Finds the set of exponential coefficients **amplitude** and **damping**, which describe the exponential curve that best represents the input data set.



This VI is a subVI of the Exponential Fit VI.

The general form of the exponential fit is given by

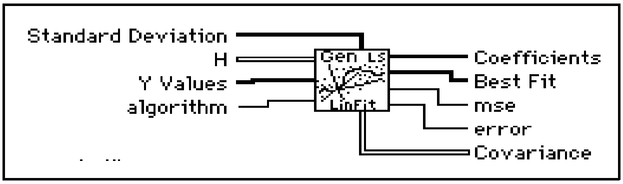
$F = a\mathrm{e}^{\tau X}$,

where F is the sequence representing the best fitted values, X represents the input sequence **X Values**, a is the **amplitude**, and τ is the **damping** constant.

General LS Linear Fit

Finds the Best Fit k-dimensional plane and the set of linear coefficients using the least chi-square method for observation data sets

$\{x_{i0}, x_{i1}, \ldots x_{ik-1}, y_i\}$ where $i = 0, 1,\ldots, n - 1$. n is the number of your observation data sets.



You can use this VI to solve multiple linear regression problems. You can also use it to solve for the linear coefficients in a multiple-function equation. Before beginning the formal description of this VI, consider both of the following, simple examples. The first example uses the General LS Linear Fit VI to perform multiple regression analysis based entirely on tabulated observation data. The second solves for the linear coefficients in a multiple-function equation.

Example 1: Predicting Cost

Suppose you want to estimate the total cost (in dollars) of a production of baked scones; using the quantity produced, *X*1, and the price of one pound of flour, X2. To keep things simple, the following five data points form this sample data table.

Cost (dollars) **Y**	**Quantity** **X1**	**Flour Price** **X2**
$150	295	3.00
$75	100	3.20
$120	200	3.10

Cost (dollars) Y	Quantity X1	Flour Price X2
$300	700	2.80
$50	60	2.50

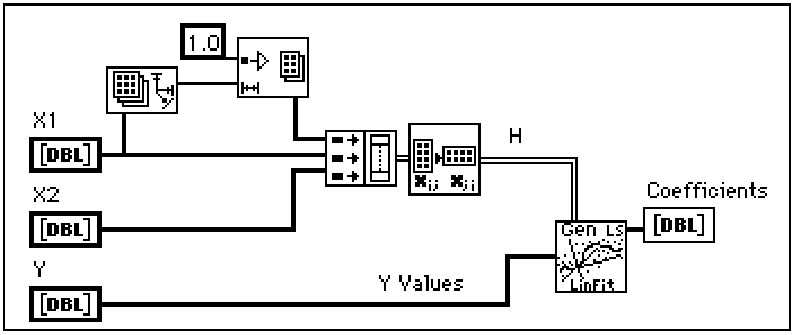
You want to estimate the coefficients to the equation:

$Y = b_0 + b_1X1 + b_2X2.$

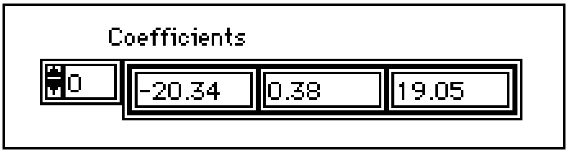
The only parameters that you need to build are **H** (observation matrix) and **Y Values**. Each column of **H** is the observed data for each independent variable: the first column is one because the coefficient b_0 is not associated with any independent variable. **H** should be filled in as:

$$H = \begin{bmatrix} 1 & 295 & 3.00 \\ 1 & 100 & 3.20 \\ 1 & 200 & 3.10 \\ 1 & 700 & 2.80 \\ 1 & 60 & 2.50 \end{bmatrix}$$

In LabVIEW, the observed data would normally appear in three arrays (*Y*, *X1*, and *X2*). The following block diagram demonstrates how to build **H** using the General LS Linear Fit VI.



After running this VI, the following coefficients are obtained.



Coefficients

0	-20.34	0.38	19.05

The resulting equation for the total cost of scone production is therefore:

$Y = -20.34 + 0.38X1 + 19.05X2$.

Example 2: Linear Combinations

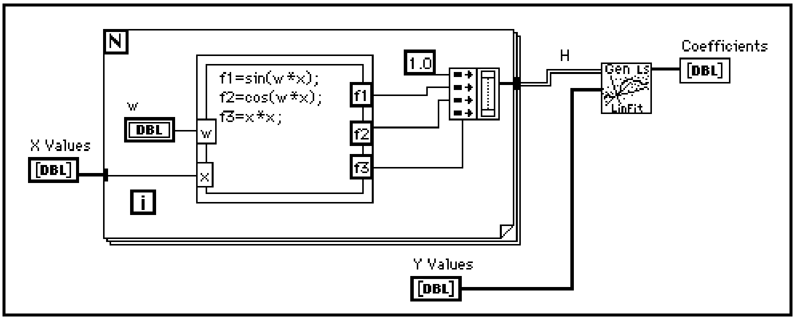
Suppose that you have collected samples from a transducer (**Y Values**) and you want to solve for the coefficients of the model:

$$y = b_o + b_1 \sin(\omega x) + b_2 \cos(\omega x) + b_3 x^2$$

To build **H**, you set each column to the independent functions evaluated at each *x* value. Assuming there are 100 *x* values, **H** is:

$$H = \begin{bmatrix} 1 & \sin(\omega x_0) & \cos(\omega x_0) & {x_0}^2 \\ 1 & \sin(\omega x_1) & \cos(\omega x_1) & {x_1}^2 \\ 1 & \sin(\omega x_2) & \cos(\omega x_2) & {x_2}^2 \\ \dots & \dots & \dots & \dots \\ 1 & \sin(\omega x_{99}) & \cos(\omega x_{99}) & {x_{99}}^2 \end{bmatrix}$$

Given that you have the independent **X Values** and observed **Y Values**, the following block diagram demonstrates how to build **H** and use the General LS Linear Fit VI.



```
f1=sin(w*x);
f2=cos(w*x);
f3=x*x;
```

The General LS Linear Fit Problem can be described as follows.

Given a set of observation data, find a set of coefficients that fit the linear “model.”

$$y_i = b_o x_{i0} + \dots + b_{k-1} x_{ik-1}$$

$$= \sum_{j=0}^{k-1} b_j x_{ij} \qquad i{=}0,\ 1,\dots,n{-}1, \tag{44-4}$$

where *B* is the set of **Coefficients**, *n* is the number of elements in **Y Values** and the number of rows of **H**, and *k* is the number of **Coefficients**.

x_{ij} is your observation data, which is contained in **H**.

$$H = \begin{bmatrix} x_{00} & x_{01}\cdots & x_{0k-1} \\ x_{10} & x_{11}\cdots & x_{1k-1} \\ . & & \\ . & & \\ . & & \\ . & & \\ x_{n-10} & x_{n-12}\cdots & x_{n-1k-1} \end{bmatrix}$$

Equation (44-3) can also be written as $Y = HB$.

This is a multiple linear regression model, which uses several variables $x_{i0}, x_{i1}, \dots, x_{ik-1}$, to predict one variable y_i. In contrast, the Linear Fit, Exponential Fit, and Polynomial Fit VIs are all based on a single predictor variable, which uses one variable to predict another variable.

In most cases, we have more observation data than coefficients. The equations in (44-4) may not have the solution. The fit problem becomes to find the coefficient *B* that minimizes the difference between the observed data, y_i and the predicted value:

$$z_i = \sum_{j=0}^{k-1} b_j x_{ij}\,.$$

This VI uses the least chi-square plane method to obtain the coefficients in (44-4), that is, finding the solution, *B*, which minimizes the quantity:

$$\chi^2 = \sum_{i=0}^{n-1} \left(\frac{y_i - z_i}{\sigma_i} \right)^2 = \sum_{i=0}^{n-1} \left(\frac{y_i - \sum_{j=0} b_j x_{ij}}{\sigma_i} \right) = |\mathrm{H_0B{-}Y_0}|^2 \tag{44-5}$$

where $h_{oij} = \frac{x_{ij}}{\sigma_i}$, $y_{oi} = \frac{y_i}{\sigma_i}$, i=0, 1,..., n–1; j=0, 1,..., k–1.

In this equation, σ_i is the **Standard Deviation**. If the measurement errors are independent and normally distributed with constant standard deviation $\sigma_i = \sigma$, the preceding equation is also the least square estimation.

There are different ways to minimize χ^2. One way to minimize χ^2 is to set the partial derivatives of χ^2 to zero with respect to b_0, b_1,..., b_{k-1}.

$$\begin{cases} \frac{\partial \chi^2}{\partial b_0} = 0 \\ \frac{\partial \chi^2}{\partial b_1} = 0 \\ . \\ . \\ . \\ . \\ \frac{\partial \chi^2}{\partial b_{k-1}} = 0 \end{cases}$$

The preceding equations can be derived to:

$$H_0^T H_0 B = H_0^T Y \tag{44-6}$$

Where H_0^T is the transpose of $\mathbf{H}_0$.

The equations in (44-6) are also called normal equations of the least-square problems. You can solve them using LU or Cholesky factorization algorithms, but the solution from the normal equations is susceptible to roundoff error.

An alternative, and preferred way to minimize χ^2 is to find the least-square solution of equations

H_0B=Y_0.

You can use QR or SVD factorization to find the solution, *B*. For QR factorization, you can choose Householder, Givens, and Givens2 (also called fast Givens).

Different algorithms can give you different precision, and in some cases, if one algorithm cannot solve the equation, perhaps another algorithm can. You can try different algorithms to find the best one based on your observation data.

The **Covariance** matrix C is computed as

$C = (H_0^T H_0)^{-1}$.

The **Best Fit** *Z* is given by

$$z_i = \sum_{j=0}^{k-1} b_j x_{ij}$$

The **mse** is obtained using the following formula:

$$\boldsymbol{mse} = \frac{1}{n} \sum_{i=0}^{n-1} \left(\frac{y_i - z_i}{\sigma_i} \right)^2$$

The polynomial fit that has a single predictor variable can be thought of as a special case of multiple regression. If the observation data sets are $\{x_i, y_i\}$ where $i = 0, 1, \ldots, n–1$, the model for polynomial fit is

$$y_i = \sum_{j=0}^{k-1} b_j x_i^j = b_0 + b_1 x_i + b_2 x_i^2 + \ldots + b_{k-1} x_i^{k-1} \qquad (44\text{-}7)$$

$i = 0, 1, 2, \ldots, n - 1$.

Comparing equations (44-4) and (44-7) shows that $x_{ij} = x_i^j$. In other words,

$$\begin{aligned} x_{i0} &= x_i^0, \quad x_{i1} = x_i, \quad x_{i2} = x_i^2, \ldots \quad x_{ik-1} = x_i^{k-1} \\ &= 1 \end{aligned}$$

In this case, you can build **H** as follows:

$$H = \left\{ \begin{array}{lllll} 1 & x_0 & x_0^2 & \dots & x_0^{k-1} \\ 1 & x_1 & x_1^2 & \dots & x_1^{k-1} \\ . & & & & \\ . & & & & \\ . & & & & \\ . & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \dots & x_{n-1}^{k-1} \end{array} \right\}$$

Instead of using $x_{ij} = x_j^i$, you can also choose another function formula to fit the data sets $\{x_i,y_i\}$. In general, you can select $x_{ij} = f_j(x_i)$. Here, $f_j(x_i)$ is the function model that you choose to fit your observation data. In polynomial fit, $f_j(x_i) = x_i^j$.

In general, you can build **H** as follows:

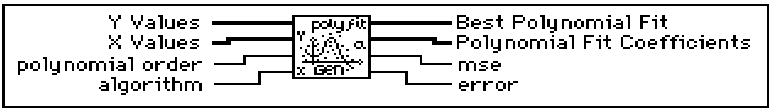
$$H = \left\{ \begin{array}{lllll} f_0(x_0) & f_1(x_0) & f_2(x_0) & \dots & f_{k-1}(x_0) \\ f_0(x_1) & f_1(x_1) & f_2(x_1) & \dots & f_{k-1}(x_1) \\ . & & & & \\ . & & & & \\ . & & & & \\ . & & & & \\ f_0(x_{n-1}) & f_1(x_{n-1}) & f_2(x_{n-1}) & \dots & f_{k-1}(x_{n-1}) \end{array} \right\}$$

Your fit model is:

$$y_i = b_0 f_0(x) + b_1 f_1(x) + \dots + b_{k-1} f_{k-1}(x) .$$

General Polynomial Fit

Finds the polynomial curve values and the set of **Polynomial Fit Coefficients**, which describe the polynomial curve that best represents the input data set.



The general form of the polynomial fit is given by

$$f_i = \sum_{j=0}^{m} a_j x_i^j$$

where *F* represents the output sequence **Best Polynomial Fit**, *X* represents the input sequence **X Values**, *A* represents the **Polynomial Fit Coefficients**, and *m* is the **polynomial order**.

The VI obtains **mse** using the formula

$$\text{mse} = \frac{1}{n} \sum_{j=0}^{n-1} (f_i - y_i)^2,$$

where *y* represents the input sequence **Y Values,** and *n* is the number of data points.

General Polynomial Fit is a special case of the General LS Linear Fit. The General Polynomial Fit VI uses the General LS Linear Fit VI as a subVI. This VI builds the *H* matrix internally using input X Values for the General LS Linear Fit VI.

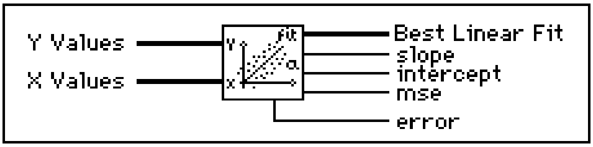
The formula used to build *H* is as follows:

$$\begin{aligned} h_{ij} &= f_j(x_i) = x_i^j \\ i &= 0, 1, \ldots, n-1 \\ j &= 0, 1, \ldots, m \end{aligned} \quad \text{For example,} \quad H = \begin{bmatrix} 1 & x_0 & \ldots . x_0^m \\ 1 & x_1 & \ldots . x_1^m \\ & & . \\ & & . \\ & & . \\ 1 & x_{n-1} & \ldots . x_{n-1}^m \end{bmatrix}$$

For more information about the General LS Linear Fit VI and the difference among different algorithms, please refer to the description of General LS Linear Fit VI.

Linear Fit

Finds the line values and the set of linear coefficients **slope** and **intercept**, which describe the line that best represents the input data set.



The general form of the linear fit is given by

$F = mX + b$,

where F represents the output sequence **Best Linear Fit**, X represents the input sequence **X Values**, m is the **slope**, and b is the **intercept**.

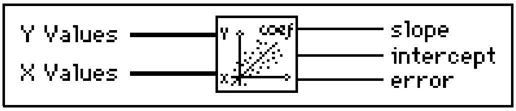
The VI obtains **mse** using the formula

$$\text{mse} = \frac{1}{n}\sum_{i=0}^{n-1}(f_i - y_i)^2 \quad ,$$

where F represents the output sequence **Best Linear Fit**, y represents the input sequence **Y Values**, and n is the number of data points.

Linear Fit Coefficients

Finds the set of linear coefficients **slope** and **intercept**, which describe the line that best represents the input data set.



This VI is a subVI of the Linear Fit VI.

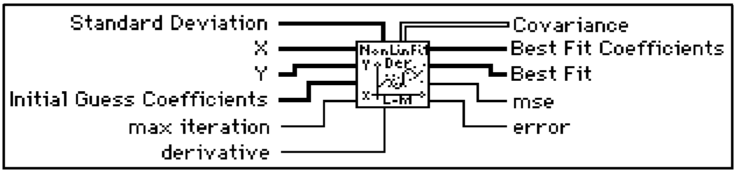
The general form of the linear fit is given by

$F = mX + b$,

where F is the sequence representing the best fitted values. X represents the input sequence **X Values**, m is the **slope**, and b is the **intercept**.

Nonlinear Lev-Mar Fit

Uses the Levenberg-Marquardt method to determine a nonlinear set of coefficients that minimize a chi-square quantity.



This VI determines the set of coefficients that minimize the chi-square quantity:

$$\chi^2 = \sum_{i=0}^{N-1}\left(\frac{y_i - f(x_i;a_1\ldots a_M)}{\sigma_i}\right)^2 \tag{44-8}$$

In this equation, (x_i, y_i) are the input data points, and $f(x_i;a_1...a_M) = f(X, A)$ is the nonlinear function where $a_1...a_M$ are coefficients. If the measurement errors are independent and normally distributed with constant, standard deviation $\sigma_i = \sigma$, this is also the least-square estimation.

You must specify the nonlinear function $f = f(X, A)$ in the Formula Node on the block diagram of the Target Fnc & Deriv NonLin VI, which is a subVI of the Nonlinear Lev-Mar Fit VI. You can access the Target Fnc & Deriv NonLin VI by selecting it from the menu that appears when you select **Project»This VI's SubVIs**.

This VI provides two ways to calculate the Jacobian (partial derivatives with respect to the coefficients) needed in the algorithm. These two methods follow:

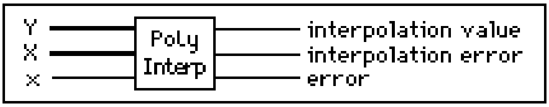
- Numerical calculation – Uses a numerical approximation to compute the Jacobian.
- Formula calculation – Uses a formula to compute the Jacobian. You need to specify the Jacobian function $\partial f/\partial A$ in the Formula Node on the block diagram of the Target Fnc & Deriv NonLin VI, as well as the nonlinear function $f = f(X, A)$. This is a more efficient computation than the numerical calculation, because it does not require a numerical approximation to the Jacobian.

The input arrays **X** and **Y** define the set of input data points. The VI assumes that you have prior knowledge of the nonlinear relationship between the x and y coordinates. That is, $f = f(X, A)$, where the set of coefficients, A, is determined by the Levenberg-Marquardt algorithm.

Using this function successfully sometimes depends on how close your initial guess coefficients are to the solution. Therefore, it is always worth taking effort and time to obtain good initial guess coefficients to the solution from any available resources before using the function.

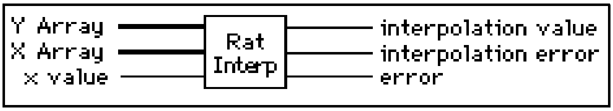
Polynomial Interpolation

Interpolates or extrapolates the function f at x, given a set of *n* points (x_i, y_i), where $f(x_i) = y_i$, *f* is any function, and given a number, *x*. The VI calculates output **interpolation value** $P_{n-1}(x)$, where P_{n-1} is the unique polynomial of degree *n*–1 that passes through the *n* points (x_i, y_i).



Rational Interpolation

Interpolates or extrapolates *f* at ***x*** using a rational function.



The rational function

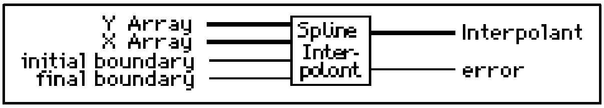
$$\frac{P(x_i)}{Q(x_i)} = \frac{p_0 + p_1 x_i + \ldots + p_m x_i^m}{q_0 + q_1 x_i + \ldots + q_v x^v}$$

passes through all the points formed by **Y Array** and **X Array**. *P* and *Q* are polynomials, and the rational function is unique, given a set of *n* points (x_i, y_i),

where $f(x_i) = y_i$, *f* is any function, and given a number *x* in the range of the x_i values. This VI calculates the output **interpolation value** y using $y = \frac{P(x)}{Q(x)}$. If the number of points is odd, the degrees of freedom of *P* and *Q* are $\frac{n-1}{2}$. If the number of points is even, the degrees of freedom of *P* are $\frac{n}{2} - 1$, and the degrees of freedom of *Q* are $\frac{n}{2}$, where *n* is the total number of points formed by **Y Array** and **X Array**.

Spline Interpolant

Returns an array **Interpolant** of length *n*, which contains the second derivatives of the spline interpolating function *g(x)* at the tabulated points x_i, where $i = 0, 1,..., n–1$. Input arrays **X Array** and **Y Array** are of length *n* and contain a tabulated function, $y_i = f(x_i)$, with $x_0<x_1<...x_{n–1}$. **initial boundary** and **final boundary** are the first derivative of the interpolating function g(x) at points 0 and *n*–1, respectively.



If **initial boundary** and **final boundary** are equal to or greater than 10^{30}, the VI sets the corresponding boundary condition for a natural spline, with zero second derivative on that boundary.

The interpolating function *g(x)* passes through all the points

$\{x_i,y_i\}$, $g(x_i) = y_i$

where $i = 0, 1,..., n–1$.

The VI obtains the interpolating function *g(x)* by interpolating every interval $[x_i, x_{i+1}]$ with a cubic polynomial function $p_i(x)$ that meets the following conditions:

- $p_i(x_i) = y_i$
- $p_i(x_{i+1}) = y_{i+1}$
- *g(x)* has continuous first and second derivatives everywhere in the range $[x_0,x_{n–1}]$:
  - $p'_i(x_i) = p'_{i+1}(x_i)$
  - $p''_i(x_i) = p''_{i+1}(x_i)$

For the preceding conditions, $i = 0, 1,..., n–2$.

From the last condition, $p''_i(x_i) = p''_{i+1}(x_i)$, we derive the following equations:

$$\frac{x_i - x_{i-1}}{6}g''(x_{i-1}) + \frac{x_{i+1} - x_{i-1}}{3}g''(x_i) + \frac{x_{i+1} - x_i}{6}g''(x_{i+1})$$

$$= \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \qquad i=1, 2,...n-2 \qquad (44\text{-}9)$$

These are *n*–2 linear equations with *n* unknowns $g''(x_i)$ $i = 0, 1, \ldots, n-1$. This VI computes $g''(x_0)$, $g''(x_{n-1})$ from **initial boundary** and **final boundary** using the formula

$$g'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + \frac{3A^2 - 1}{6}(x_{i+1} - x_i)g''(x_i)$$

$$+ \frac{3B^2 - 1}{6}(x_{i+1} - x_i)g''(x_{i+1}) \quad .$$

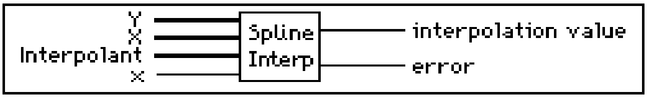
Here

$$A = \frac{x_{i+1} - x}{x_{i+1} - x_i} \qquad B = 1 - A = \frac{x - x_i}{x_{i+1} - x_i}$$

You can derive this formula from the preceding conditions numbered 1–3. This VI then uses $g''(x_0)$, $g''(x_{n-1})$ in equation (44-1) to solve all the $g''(x_i)$, for *i* = 1, … *n*–2. $g''(x_i)$ is the output **Interpolant**. You can use **Interpolant** as an input to the Spline Interpolation VI to interpolate *y* at any value of $x_0 \le x \le x_{n-1}$.

Spline Interpolation

Performs a cubic spline interpolation of *f* at *x*, given a tabulated function.



This VI performs cubic spline interpolation using a tabulated function in the form of $y_i = f(x_i)$ for *i* = 0, 1,..., *n*–1, and given the second derivatives **Interpolant** that the VI obtains from the Spline Interpolant VI. The value of **x** must be in the range of **X** values. The points are formed by the input arrays **X** and **Y**, and *n* is the total number of points.

On the interval $[x_i, x_{i+1}]$, the output **interpolation value** y is defined by

$$y = Ay_i + By_i + 1 + Cy''_i + Dy''_i + 1 \ ,$$

and

$$A = \frac{x_{i+1} - x}{x_{i+1} - x_i} ,$$

$$B = 1 - A,$$

$$C = \frac{1}{6}(A^3 - A)(x_{i+1} - x_i)^2 ,$$

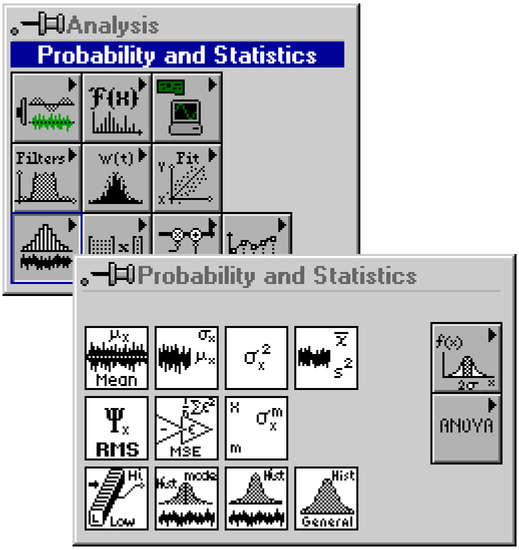
$$D = \frac{1}{6}(B^3 - B)(x_{i+1} - x_i)^2 .$$

Analysis Probability and Statistics VIs

Chapter 45

This chapter describes the VIs that perform probability, descriptive statistics, analysis of variance, and interpolation functions.

To access the **Probability and Statistics** palette, choose **Functions»Analysis»Probability and Statistics**, as shown in the following illustration.



For examples of how to use the statistics VIs, see the examples located in `examples\analysis\statxmpl.llb`.

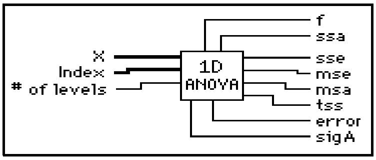
☞ **Note:** ***These VIs are not available in the Base Analysis package.***

Probability and Statistics VI Descriptions

The following Probability and Statistic VIs are available.

1D ANOVA

Takes an array, **X**, of experimental observations made at various **levels** of a factor, with at least one observation per level, and performs a one-way analysis of variance in the fixed effect model. In the one-way analysis of variance, the VI tests whether the level of the factor has an effect on the experimental outcome.



Factors and Levels

A factor is a basis for categorizing data. For example, if you count the number of sit-ups individuals can do, one basis of categorization is age. For age, you might have the following levels:

Level 0:	6 years old to 10 years old
Level 1:	11 years old to 15 years old
Level 2:	16 years old to 20 years old

Now, suppose that you make a series of observations to see how many sit-ups people can do. If you take a random sampling of five people, you might find the following results:

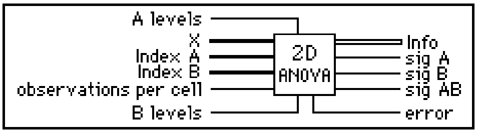
Person 1	8 years old (level 0)	10 sit-ups
Person 2	12 years old (level 1)	15 sit-ups
Person 3	16 years old (level 2)	20 sit-ups
Person 4	20 years old (level 2)	25 sit-ups
Person 5	13 years old (level 1)	17 sit-ups

Notice that you have made at least one observation per level. To perform an analysis of variance, you must make at least one observation per level.

To perform the analysis of variance, you specify an array **X** of observations, with values 10, 15, 20, 25, and 17. The array **Index** specifies the level (or category) to which each observation applies. In this case, **Index** has the values 0, 1, 2, 2, and 1. Finally, there are three possible levels, so you pass in a value of 3 for the **# of levels** parameter.

2D ANOVA

Takes an array of experimental observations made at various levels of two factors and performs a two-way analysis of variance.



Factors, Levels, and Cells

A factor is a basis for categorizing data. For example, if you count the number of sit-ups individuals can do, one basis of categorization is age. For age, you might have the following levels:

Level 0: 6 years old to 10 years old

Level 1: 11 years old to 15 years old

Another possible factor is weight, with the following levels:

Level 0: less than 50 kg

Level 1: between 50 and 75 kg

Level 2: more than 75 kg

Now, suppose that you made a series of observations to see how many sit-ups people could do. If you took a random sampling of *n* people, you might find the following results:

Person 1	8 years old (level 0)	30 kg (level 0)	10 sit-ups
Person 2	12 years old (level 1)	40 kg (level 0)	15 sit-ups
Person 3	15 years old (level 1)7	6 kg (level 2)	20 sit-ups
Person 4	14 years old (level 1)	60 kg (level 1)	25 sit-ups
Person 5	9 years old (level 0)	51 kg (level 1)	17 sit-ups
Person 6	10 years old (level 0)	80 kg (level 2)	4 sit ups

and so on.

If you plot observations as a function of factor A and factor B, they fall into cells of a matrix with factor A as rows and factor B as columns. Each cell must contain at least one observation, and each cell must contain the same number of observations.

To perform the analysis of variance, you specify an array **X** of observations, with values 10, 15, 20, 25, 17, and 4. The array **Index A** specifies the level (or category) of factor A to which each observation applies. In this case, the array would have the values 0, 1, 1, 1, 0, and 0.

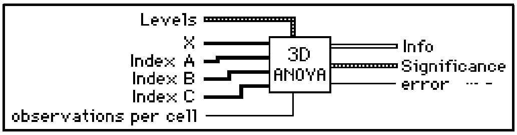
The array **Index B** specifies the level (or category) of factor B to which each observation applies. In this case, the array would have the values 0, 0, 2, 1, 1, and 2. Finally, there are two possible levels for factor A and three possible levels for factor B, so you pass in a value of 2 for the **A levels** parameter, and a value of 3 for the **B levels** parameter.

You can apply any one of the following models, where L is the specified **observations per cell**:

- Model 1: Fixed-effects with no interaction and one observation per cell (per specified levels *x* and *y* of the factors A and B, respectively).
- Model 2: Fixed-effects with interaction and L>1 observations per cell.
- Model 3: Either of the mixed-effects models with interaction and L>1 observations per cell.
- Model 4: Random-effects with interaction and L>1 observations per cell.

3D ANOVA

Takes an array of experimental observations made at various levels of three factors and performs a three-way analysis of variance. In any ANOVA, you look for evidence that the factors or interactions among factors have a significant effect on experimental outcomes. What varies with each model is the method used to do this.



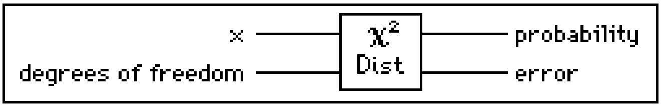
The three-way ANOVA models are as follows, where L is the number of **observations** per cell:

- Fixed-effects with interaction and L>1 observations per cell.
- Any of the six mixed-effects models with interaction and L>1 observations per cell.
- Random-effects with interaction and L>1 observations per cell.

A factor is a basis for categorizing data. A cell of data consists of all those experimental observations that fall in particular levels of the three factors. The number of observations that fall in a cell must be some constant number L, which does not vary between cells. See the description of factors, levels, and cells in the 2D ANOVA VI description. Remember that a cell in this 3D ANOVA VI is the intersection of three factors instead of two as described in the 2D ANOVA VI description.

Chi Square Distribution

Computes the one-sided **probability**, p, of the χ^2 distributed random variable, **x**, with the specified **degrees of freedom**.

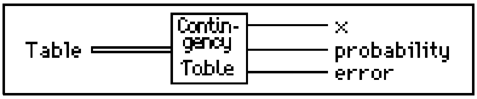


p = Prob {X $\leq$ **x**},

where X is χ^2 distributed with n **degrees of freedom**, p is the **probability**, n is **degrees of freedom**, and **x** is the value.

Contingency Table

Classifies and tallies objects of experimentation according to two schemes of categorization.



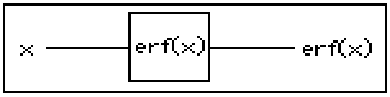
With the χ^2 test of homogeneity, the VI takes a random sample of some fixed size from each of the categories in one categorization scheme. For each of the samples, the VI categorizes the objects of experimentation according to the second scheme, and tallies them. The VI tests the hypothesis to determine whether the populations from which each sample is taken are identically distributed with respect to the second categorization scheme.

With the χ^2 test of independence, the VI takes only one sample from the total population. The VI then categorizes each object and tallies it in two categorization schemes. The VI tests the hypothesis that the categorization schemes are independent.

You must choose a level of significance for each test. This is how likely you want it to be that the VI rejects the hypothesis when it is true. Ordinarily, you do not want it to be very likely. So you should use a small number (0.05 or 5 percent is a common choice) to determine the level of significance. The output parameter **probability** is the level of significance at which the hypothesis is rejected. Thus, if **probability** is less than the level of significance, you must reject the hypothesis.

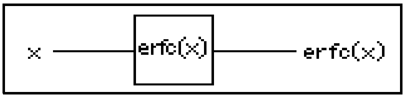
erf(x)

Evaluates the error function at the input value.



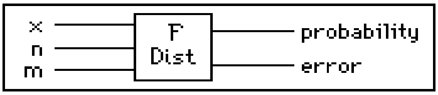
erfc(x)

Evaluates the complementary error function at the input value.



F Distribution

Computes the one-sided **probability**, p, of the F-distributed random variable, F, with the specified **n** and **m** degrees of freedom

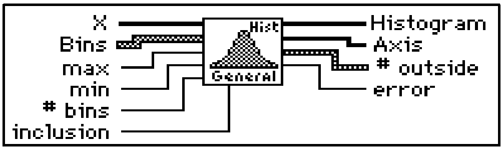


p = Prob $\{F_{n,m} \leq \mathbf{x}\}$,

where F is F-distributed, p is the **probability**, **n** specifies the first degree of freedom, **m** specifies the second degree of freedom, and **x** is the value.

General Histogram

Finds the discrete histogram of the input sequence **X** based on the given bin specifications.



The VI obtains the **Histogram** as follows. The VI establishes all the intervals (also called bins) based on the information in the input array **Bins** first. The intervals (bins) are:

$_{\Delta i}$ = (**Bins**[*i*].lower: **Bins**[*i*].upper) $\quad i = 0, 1, 2, ..., k-1$

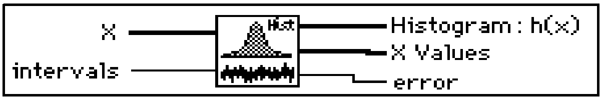
where

Bins[*i*].lower is the value **lower** in the *i*th cluster of array **Bins**, **Bins**[*i*].upper is the value **upper** in the *i*th cluster of array **Bins**, *k* is the number of elements in **Bins**, which consists of the number of total intervals (bins).

Whether the two ending points **Bins**[*i*].lower and **Bins**[*i*].upper of each interval (bin) are included in the interval (bin) $_{\Delta i}$ depends on the value of **bin inclusion** in the corresponding cluster *i* of the **Bins**.

Histogram

Finds the discrete histogram of the input sequence **X**. The histogram is a frequency count of the number of times that a specified interval occurs in the input sequence.



If the input sequence is

X = {0, 1, 3, 3, 4, 4, 4, 5, 5, 8},

then the **Histogram: h(x)** of **X** for eight **intervals** is

$h(X) = \{h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7\} = \{1, 1, 0, 2, 3, 2, 0, 1\}$.

Notice that the histogram of the input sequence **X** is a function of **X**.

The VI obtains **Histogram: h(x)** as follows. The VI scans the input sequence **X** to determine the range of values in it. Then the VI establishes the interval width, ý*x*, according to the specified number of **intervals**,

$$\Delta x = \frac{\max - \min}{m},$$

where max is the maximum value found in the input sequence **X**, min is the minimum value found in the input sequence **X**, and *m* is the specified number of **intervals**.

Let χ represent the output sequence **X Values**, because the histogram is a function of **X**. The VI evaluates elements of χ using

$$\chi_i = \min + 0.5\Delta x + i\Delta x \text{ for } i = 0, 1, 2, \ldots, m-1.$$

The VI defines the i^{th} interval Δ_i to be the range of values from $\chi_i - 0.5\ \Delta x$ up to but not including $\chi_i + 0.5\ \Delta x$,

$\Delta_i = [\chi_i - 0.5\ \Delta x : \chi_i + 0.5\ \Delta x)$, for $i = 0, 1, 2,\ldots, m-1$,

and defines the function $y_i(x)$ to be

$$y_i(x) = \begin{cases} 1 & \text{if} x \in \supseteq \Delta_i \\ 0 & \text{elsewhere} \end{cases} .$$

The function has unity value if the value of x falls within the specified interval. Otherwise it is zero. Notice that the interval Δ_i is centered about χ_i, and its width is Δ_x.

The last interval, Δ_{m-1}, is defined as $[\chi_{m-\iota} - 0.5\Delta x : \chi_{m-\iota} + 0.5\Delta x]$. In other words, if a value is equal to max, it is counted as belonging to the last interval.

Finally, the VI evaluates the histogram sequence H using

$$h_i = \sum_{j=0}^{n-1} y_i(x_j) \text{ for } i = 0, 1, 2,\ldots, m-1,$$

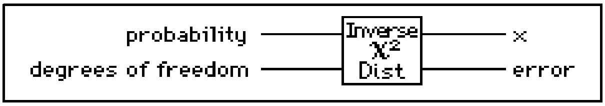
where h_i represents the elements of the output sequence **Histogram: h(x)**, and n is the number of elements in the input sequence **X**.

Inv Chi Square Distribution

Computes the value of **x** such that the condition

p = *Prob* {X ≤ **x**}

is satisfied, given the **probability** value, p, of a X^2 -distributed random variable, X, with n **degrees of freedom**.

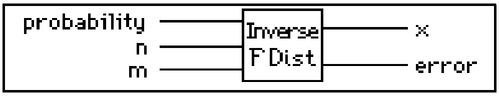


Inv F Distribution

Computes the value of **x** such that the condition

$p = \{Prob_{n,m} \le X\}$

is satisfied, given the **probability** value *p* of an F-distributed random variable, *F*, with **n** and **m** degrees of freedom.

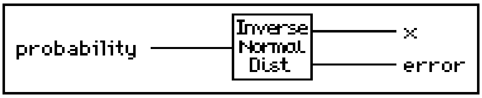


Inv Normal Distribution

Computes the value of **x** such that the condition

p = Prob {X ≤ **x**}

is satisfied, given the **probability** value, *p*, of a Normally distributed random variable, *X*.

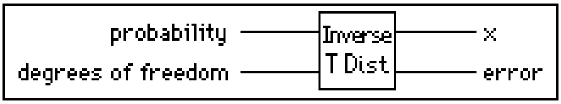


Inv T Distribution

Computes the value of **x** such that the condition

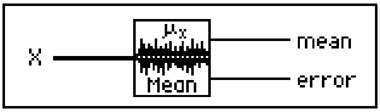
p = Prob {T_n ≤ **x**}

is satisfied, given the **probability** value, *p*, of a T-distributed random variable, *T*, with n **degrees of freedom**.



Mean

Computes the mean (average) of the values in the input sequence **X**.



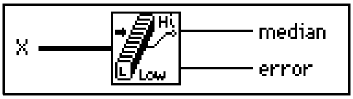
This VI computes **mean** (µ) using the following formula:

$$\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i ,$$

where n is the number of elements in **X**.

Median

Finds the median value of the input sequence **X** by sorting the values of **X** and selecting the middle element(s) of the sorted array.



Let n be the number of elements in the input sequence **X,** and let S be the sorted sequence of **X**. The VI finds **median** using the following identity:

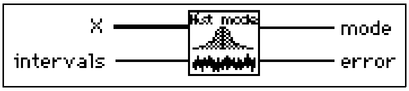
$$\text{median} = \begin{cases} s_i & \text{if} n \text{ is odd} \\ 0.5(s_{k-1} + s_k) & \text{if} n \text{ is even} \end{cases}$$

where $i = \frac{n-1}{2}$,

and $k = \frac{n}{2}$.

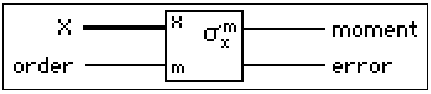
Mode

Finds the **mode** of the input sequence **X**.



Moment About Mean

Computes the moment about the mean of the input sequence **X** using the specified **order**.



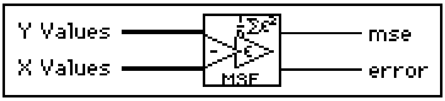
Let m be the desired **order**. The VI computes the m^{th}-order **moment** using the formula:

$$\sigma_x^m = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - \mu)^m,$$

where σ_x^m is the m^{th}-order **moment**, and n is the number of elements in the input sequence **X**.

MSE

Computes the mean squared error (**mse**) of the input sequences **X Values** and **Y Values**.



The VI uses the following formula to find **mse**:

$$\text{mse} = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - y_i)^2,$$

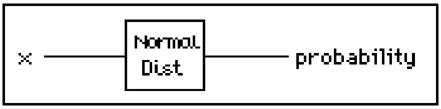
where n is the number of data points.

Normal Distribution

Computes the one-sided **probability**, p, of the normally distributed random variable, **x**,

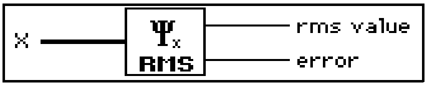
$p = \text{Prob}\ \{X \le \mathbf{x}\}$,

where X is standard Normally distributed, p is the **probability**, and **x** is the value.



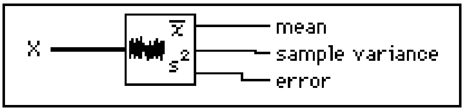
RMS

Computes the root mean square (rms) of the input sequence **X**.



Sample Variance

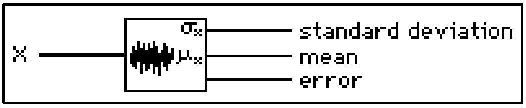
Computes the **mean** and **sample variance** of the values in the input sequence *X*.



☞ **Note:** ***If you need to compute the sample standard deviation of* X*, take the square root of* sample variance.**

Standard Deviation

Computes the mean value and the standard deviation of the values in the input sequence **X**.



This VI computes **standard deviation** (σ_x) and **mean** (μ) using the following formula:

$$\sigma_x = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(x_i - \mu)^2},$$

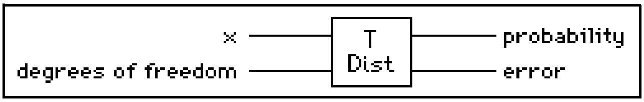
where $\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i$, and n is the number of elements in **X**.

T Distribution

Computes the one-sided **probability**, p, of the t-distributed random variable, T_n, with the specified **degrees of freedom**

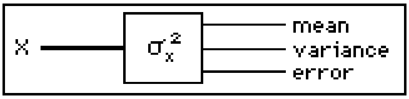
p = Prob $\{T_n \leq \mathbf{x}\}$,

where *T* is t-distributed, *p* is the **probability**, *n* is **degrees of freedom**, and **x** is the value.



Variance

Computes the variance and the mean value of the input sequence **X**.



This VI computes **variance** ($\sigma_x{}^2$) and **mean** (μ) using the following formula:

$$\sigma_x^2 = \frac{1}{n}\sum_{i=0}^{n-1}(x_i - \mu)^2,$$

where $\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i$, and n is the number of elements in **X**.

Chapter 46

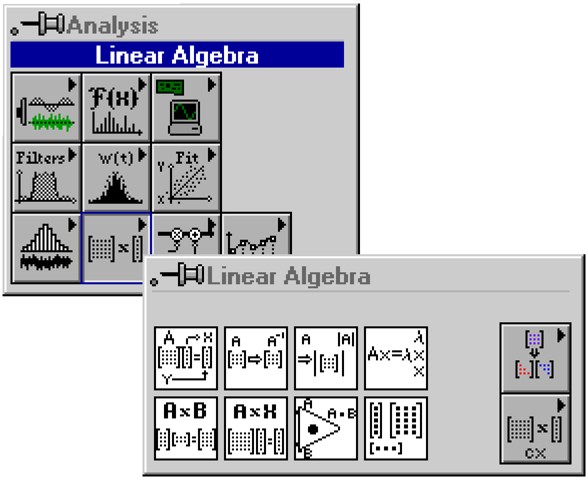
Analysis Linear Algebra VIs

This chapter describes the VIs that perform matrix related computation and analysis, and provides overviews on the following:

- Basic Matrix Manipulations
- Solving Linear Equations and Matrix Inverses
- Eigenvalues and Eigenvectors
- Matrix Analysis

It includes both real and complex matrices.

To access the **Linear Algebra** palette, choose **Functions»Analysis»Linear Algebra**, as shown in the following illustration.



This chapter is divided into the following groups:

- Matrix factorization
- Solving linear equations and matrix inverses
- Eigenvalues and Eigenvectors problems
- Matrix analysis

A matrix is represented by a 2D array:

$$A = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0n-1} \\ a_{10} & a_{11} & \cdots & a_{1n-1} \\ \cdots & \cdots & \cdots & \cdots \\ a_{m-10} & a_{m-11} & \cdots & a_{m-1n-1} \end{bmatrix}$$

A is an *m*-by-*n* matrix that contains *m* rows and *n* columns.

A matrix is called a rectangular matrix in general. When *m*=*n*, it is called a square matrix.

Basic Matrix Manipulations Functions

This section provides an overview of Basic Matrix Manipulations.

Addition

$C = A + B \Rightarrow c_{ij} = a_{ij} + b_{ij}$

A, **B**, and **C** have the same dimension size.

Matrix-Matrix Multiplication

$$C = AB \Rightarrow c_{ij} = \sum_{k=0}^{r-1} a_{ik}b_{kj}$$

If **A** is a *n*-by-*r* matrix, and **B** is a *r*-by-*m* matrix, then **C** is a *n*-by-*m* matrix.

Scalar-Matrix Multiplication

$C = \alpha A \Rightarrow c_{ij} = \alpha a_{ij}$

C and **A** have the same dimension size.

Transposition

For a real matrix:

$$C = A^T \Rightarrow c_{ij} = a_{ji}$$

For a complex matrix, it is the complex conjugate transposition:

$$C = A^H \Rightarrow c_{ij} = a^*_{ji}$$

Complex conjugate: if $a = x + iy$, then conjugate $a^* = x - iy$. If **A** is an *m*-by-*n* matrix, then **C** is an *n*-by-*m* matrix and is called the transpose of **A**.

Common Matrices

This section describes the Common Matrices.

Identity Matrix

$$A = \begin{bmatrix} 1\ 0\ \dots\ 0 \\ 0\ 1\ \dots\ 0 \\ \\ 0\ 0\ \dots\ 1 \end{bmatrix}, \ a_{ij} = 0 \text{ when } i \neq j,\ a_{ij} = 1 \text{ when } i = j.$$

A is a square matrix.

Diagonal Matrix

$$A = \begin{bmatrix} a_{00} & 0 & \dots & 0 \\ 0 & a_{11} & \dots & 0 \\ \\ 0 & 0 & \dots & a_{m-1n-1} \end{bmatrix}, \ a_{ij} = 0 \text{ when } i \neq j.$$

Hermitian Matrix

If a complex matrix **A** satisfies $A = A^H$, **A** is called a Hermitian matrix.

Symmetric Matrix

Matrix **A** is called a symmetric matrix if $a_{ij} = a_{ji}$, that is $A = A^T$.

Upper Triangular Matrix

$$A = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0n-1} \\ 0 & a_{11} & \cdots & a_{1n-1} \\ & & & \\ 0 & 0 & \ldots & a_{m-1n-1} \end{bmatrix}, \; a_{ij} = 0 \text{ when } i>j.$$

Lower Triangular Matrix

$$A = \begin{bmatrix} a_{00} & 0 & \ldots & 0 \\ a_{10} & a_{11} & \ldots & 0 \\ & & & \\ a_{m-10} & a_{m-11} & \cdots & a_{m-1n-1} \end{bmatrix}, \; a_{ij} = 0 \text{ when } i<j.$$

Orthogonal Matrix

Matrix **A** is called orthogonal if $A^T A = I$, and **I** is an identity matrix.

Permutation Matrix

A permutation matrix is an identity matrix with some rows or columns exchanged. A permutation matrix is an orthogonal matrix.

Positive Definite Matrix

A real matrix is positive definite if and only if it is symmetric; that is, $A = A^T$, and the quadratic form $X^T AX > 0$ for all nonzero vectors *X*.

A complex matrix is positive definite if and only if it is Hermitian; that is, $A = A^H$ and $X^H AX > 0$ for all nonzero, complex vectors *X*.

Matrix Factorization

A matrix can be factored into the multiplication of several, simpler matrices. You can use these factored, simple matrices to solve some matrix problems, such as solving a linear equation, inverting a matrix, and finding the determinant of a matrix.

The common factorization methods include LU, Cholesky, QR, and Singular Value Decomposition (SVD).

- LU Factorization—Factors a square matrix into two matrices. One is an upper triangular matrix **U**, and the other is a lower triangular matrix **L** that has ones on the diagonal, so that *PA*=*LU*. **P** is a permutation matrix.
- When a square matrix is positive definite, you can factor it into $A = R^T R$, if *A* is a real matrix, and $A = R^H R$, if *A* is a complex matrix, where R is an upper triangular matrix. This is called Cholesky factorization. Cholesky factorization only needs half of the operations of LU factorization.
- QR Factorization–Factors a matrix as the product of an orthogonal matrix **Q** and an upper triangular matrix **R**: *A*=*QR*. QR factorization is useful for both square and rectangular matrices.
- SVD—Decomposes a matrix into the product of three matrices: $A = USV^T$, where *U* and *V* are orthogonal matrices and *S* is a diagonal matrix whose diagonal values are called the singular values of *A*. SVD is useful for solving analysis problems involving matrices. In addition to its common uses, you can use SVD for operations such as pseudoinverse, rank, norm, and condition number.

Solving Linear Equations and Matrix Inverses

To Solve the linear equation *AX*=*Y*, you must find solution *X* when you know the given values of *A* and *Y*. *A* is a *m*-by-*n* matrix, *X* is a vector with *n* elements, and *Y* is a vector with *m* elements.

Using LU factorization, if *m*=*n* and **A** is a square matrix, **A** can be factored into triangular matrices **L** and **U**, so that *A*=*LU*. *AX*=*Y* becomes *LUX*=*Y* and you can solve *Z* for *LZ*=*Y* where *Z*=*UX*. You can then solve for *X* in *UX*=*Z*.

In the Cholesky case, $L = R^T$ and $U = R$.

Triangular systems are easy to solve using recursive techniques.

If $m \neq n$, the number of equations are different from the number of unknowns and **A** is not a square matrix, **A** can be factored into an orthogonal matrix **Q** and an upper triangular matrix **R**, so that *A*=*QR*. *AX*=*Y* becomes *QRX*=*Y* and you can solve for *X* by using $RX = Q^T Y$.

When *m*>*n*, and the system has more equations than unknowns, it is called an overdetermined system. The solution that satisfies *AX*=*Y* may not exist. The solution above finds the least square solution that minimizes $\|AX - Y\| = \sum[(AX)_i - y_i]^2$.

When $m<n$, and the system has more unknowns than equations, it is called an underdetermined system. It may have infinite solutions that satisfy $AX=Y$. The previous solution finds one of these solutions.

Inverting a square matrix **A** means that you find A^{-1} that satisfies $AA^{-1} = I$, where I is an identity matrix. A^{-1} is called the inverse of matrix **A**. You can solve for A^{-1} by solving n linear equations $AA^{-1} = I$.

When **A** is not a square matrix, or when **A** is singular, A^{-1} does not exist. You can compute the pseudoinverse of A instead. If the m-by-n matrix $\mathbf{A}^+$ satisfies the following four Moore-Penrose conditions:
$AA^+A=A$
$A^+AA^+=A^+$
AA^+ is a Hermitian matrix if A is a complex matrix. AA^+ is a symmetric matrix if A is real matrix.
A^+A is a Hermitian matrix if A is a complex matrix. A^+A is a symmetric matrix if A is real matrix.
Then, A^+ is called the pseudoinverse of matrix **A**. You can compute for A^+ using SVD.

Eigenvalues and Eigenvectors

This Eigenvalue problem is to determine the nontrivial solutions to the equation $AX = \lambda X$, where A is an n-by-n matrix, X is a vector with n elements, and λ is a scalar. The n values of λ that satisfy the equation are called eigenvalues of A, and the corresponding values of X are called the right eigenvectors of A.

Matrix Analysis

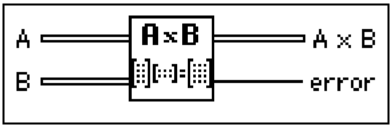
Matrix Analysis VIs can compute the matrix determinant, condition number, norm, and rank. Typically, you use these parameters to analyze a matrix property.

Linear Algebra VI Descriptions

The following Linear Algebra VIs are available.

A x B

Performs the matrix multiplication of two input matrices.



If *A* is an *n*-by-*k* matrix and *B* is a *k*-by-*m* matrix, the matrix multiplication of *A* and *B*, *C* = *AB,* results in a matrix, *C*, whose dimensions are *n*-by-*m*. Let *A* represent the 2D input array **A** matrix, *B* represent the 2D input array **B** matrix, and *C* represent the 2D output array **A x B**. The VI obtains the elements of *C* using the formula

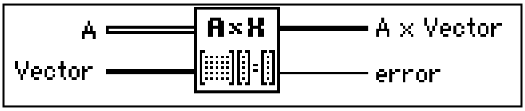
$$c_{ij} = \sum_{l=0}^{k-1} a_{il} b_{lj} \qquad \text{for} \begin{cases} i = 0, 1, 2, ..., n-1 \\ j = 0, 1, 2, ..., m-1 \end{cases},$$

where *n* is the number of rows in **A** matrix, *k* is the number of columns in **A** matrix and the number of rows in **B** matrix, and *m* is the number of columns in **B** matrix.

Note: ***The A x B VI performs a strict matrix multiplication and not an element-by-element 2D multiplication. To perform an element-by-element multiplication, you must use the LabVIEW Multiply function. In general, AB ¦ BA.***

A x Vector

Performs the multiplication of an input matrix and an input vector.



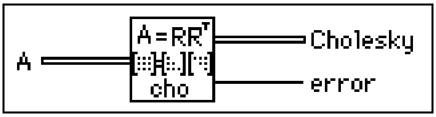
If **A** is an *n*-by-*k* matrix, and *X* is a vector with *k* elements, the multiplication of **A** and *X*, *Y* = *AX,* results in a vector *Y* with *n* elements. Let *Y* represent the output **A x Vector**. The VI obtains the elements of *Y* using the formula

$$y_i = \sum_{j=0}^{k-1} a_{ij} x_j \text{ , for } i = 0, 1, 2, \ldots, n–1,$$

where *n* is the number of rows in **A**, and *k* is the number of columns in **A** and the number of elements in *X*.

Cholesky Factorization

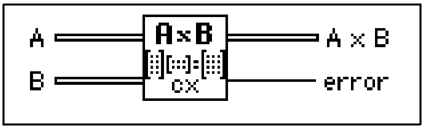
Performs Cholesky factorization for a real, positive definite matrix **A**.



If the real, square matrix **A** is positive definite, you can factor it as $A = R^T R$, where R is an upper triangular matrix, and R^T is the transpose of R.

Complex A x B

Performs the matrix multiplication of two input complex matrices.



If *A* is an *n*-by-*k* matrix and *B* is a *k*-by-*m* matrix, the matrix multiplication of *A* and *B*, *C* = *AB,* results in a matrix, *C*, whose dimensions are *n*-by-*m*. Let *A* represent the 2D input array **A** matrix, *B* represent the 2D input array **B** matrix, and *C* represent the 2D output array **A x B**. The VI obtains the elements of *C* using the formula

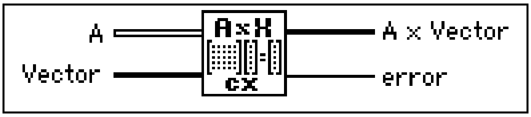
$$c_{ij} = \sum_{l=0}^{k-1} a_{il} b_{lj} \qquad \text{for} \begin{cases} i = 0, 1, 2, \ldots, n-1 \\ j = 0, 1, 2, \ldots, m-1 \end{cases},$$

where *n* is the number of rows in **A** matrix, *k* is the number of columns in **A** matrix and the number of rows in **B** matrix, and *m* is the number of columns in **B** matrix.

Note: ***The Complex A x B VI performs a strict matrix multiplication and not an element-by-element 2D multiplication. To perform an element-by-element multiplication, you must use the LabVIEW Multiply function. In general,*** $AB \neq BA$.

Complex A x Vector

Performs the multiplication of a complex input matrix and a complex input vector.



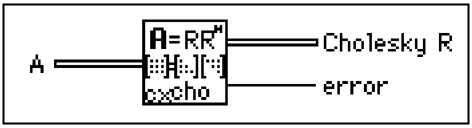
If **A** is an *n*-by-*k* matrix, and *X* is a vector with *k* elements, the multiplication of **A** and *X*, $Y = AX$, results in a vector *Y* with *n* elements. Let *Y* represent the output **A x Vector,** *X* represents the input vector. The VI obtains the elements of *Y* using the formula

$$y_i = \sum_{j=0}^{k-1} a_{ij}x_j \text{ , for } i = 0, 1, 2, \ldots, n{-}1,$$

where *n* is the number of rows in **A**, and *k* is the number of columns in **A** and the number of elements in *X*.

Complex Cholesky Factorization

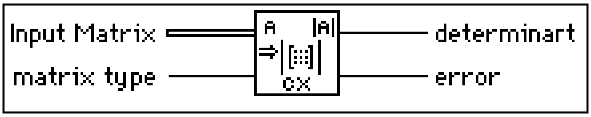
Performs Cholesky factorization of a complex, positive definite matrix **A**.



If the complex square matrix **A** is positive definite, it can be factored as $A = R^H R$, where *R* is an upper triangular matrix and R^H is the complex conjugate transpose of *R*.

Complex Determinant

Finds the **determinant** of a complex, square matrix **Input Matrix**.



Let *A* denote a square matrix that represents the **Input Matrix,** and let *L* and *U* be the lower and upper triangular matrices, respective, of *A* such that

$A = LU$,

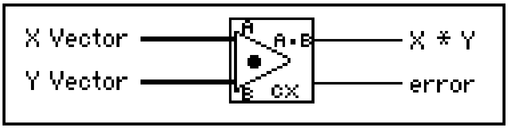
where the main diagonal elements of the lower triangular matrix L are arbitrarily set to one. The VI finds the **determinant** of A by the product of the main diagonal elements of the upper triangular matrix U:

$$|A| = \prod_{i=0}^{n-1} u_{ii} ,$$

where $|A|$ is the **determinant** of **A**, and n is the dimension of **A**.

Complex Dot Product

Computes the dot product of complex **X Vector** and **Y Vector**.



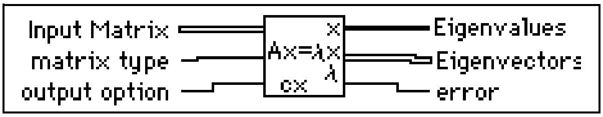
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains the dot product **X*Y** using the formula:

$$X^*Y = \sum_{i=0}^{n-1} x_i y_i \ ,$$

where n is the number of data points. Notice that the output value **X*Y** is a complex scalar value.

Complex Eigenvalues & Vectors

Finds the **Eigenvalues** and right **Eigenvectors** of a square complex **Input Matrix A**.



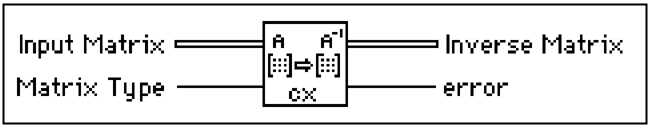
The eigenvalue problem is to determine the nontrivial solutions for the equation:

$AX = \lambda X$

where A represents an n-by-n **Input Matrix**, X represents a vector with n elements, and λ is a scalar. The n values of λ that satisfy the equation are the **Eigenvalues** of A and the corresponding values of X are the right **E**igenvectors of A. A Hermitian matrix always has real eigenvalues.

Complex Inverse Matrix

Finds the **Inverse Matrix** of a complex matrix **Input Matrix**.



Let A be the **Input Matrix** and I be the identity matrix. You obtain the **Inverse Matrix** by solving the system $AB = I$ for B.

If A is a nonsingular matrix, you can show that the solution to the preceding system is unique and that it corresponds to the inverse matrix of A

$B = A^{-1}$,

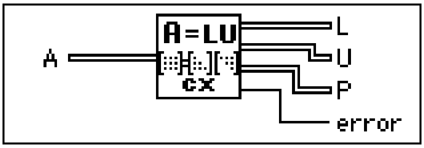
and B is therefore the **Inverse Matrix**. A nonsingular matrix is a matrix in which no row or column contains a linear combination of any other row or column, respectively.

☞ **Note:** ***You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Complex Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.***

The numerical implementation of the matrix inversion is not only numerically intensive but, because of its recursive nature, it is also highly sensitive to round-off error introduced by the floating point, numeric coprocessor. Although the computations use the maximum possible accuracy, the VI cannot always solve for the system.

Complex LU Factorization

Performs the LU factorization of a complex, square matrix **A**.



LU factorization factors the square matrix **A** into two triangular matrices; one is a lower triangular matrix **L** with ones on the diagonal, and the other is an upper triangular matrix **U**, so that

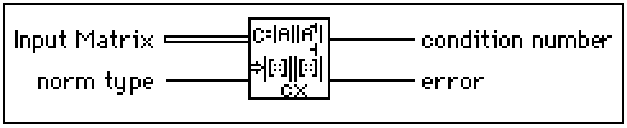
$$PA = LU$$

where **P** is a permutation matrix, which consists of the identity matrix with some rows exchanged.

Factorization is the key step for inverting a matrix, computing the determinant of a matrix, and solving a linear equation.

Complex Matrix Condition Number

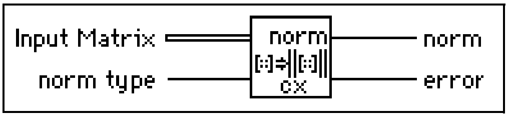
Computes the **condition number** of a complex matrix **Input Matrix**.



The **condition number** of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from the matrix inversion and linear equation solutions.

Complex Matrix Norm

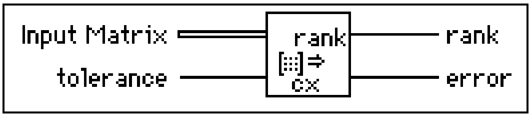
Computes the **norm** of a complex matrix **Input Matrix**.



The **norm** of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. Let **A** represent the **Input Matrix**, $\|A\|_p$ represent the **norm** of A, where p can be $1,2,f,\infty$. Different values of p mean different types of norms that are computed.

Complex Matrix Rank

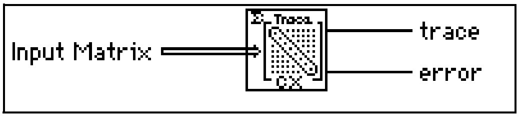
Computes the **rank** of a rectangular, complex matrix **Input Matrix**.



rank is the number of singular values of the **Input Matrix** that are larger than the **tolerance**. **rank** is the maximum number of independent rows or columns of the **Input Matrix**.

Complex Matrix Trace

Finds the **trace** of **Input Matrix**.



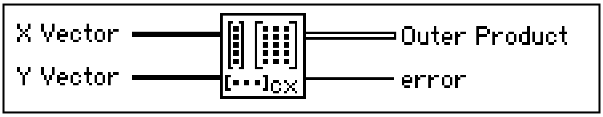
Let *A* be a square matrix that represents **Input Matrix** and tr(*A*) be **trace**. The **trace** of A is the sum of the main diagonal elements of *A*

$$\operatorname{tr}(A) = \sum_{i=0}^{n-1} a_{ii} ,$$

where *n* is the dimension of **Input Matrix**.

Complex Outer Product

Computes the outer product of a complex **X Vector** and **Y Vector.**



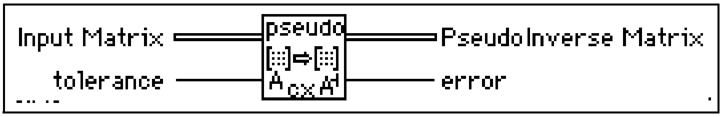
Let *X* represent the input sequence **X Vector** and *Y* represent the input sequence **Y Vector**. The VI obtains **Outer Product** using the formula:

$$a_{ij} = x_i\, y_j, \text{ for } \begin{cases} i = 0, 1, 2, \ldots, n-1 \\ j = 0, 1, 2, \ldots, m-1 \end{cases},$$

where *A* represents the 2D output sequence **Outer Product**, *n* is the number of elements in the input sequence **X Vector**, and *m* is the number of elements in the input sequence **Y Vector**.

Complex PseudoInverse Matrix

Finds the PseudoInverse Matrix of a rectangular, complex matrix **Input Matrix**.

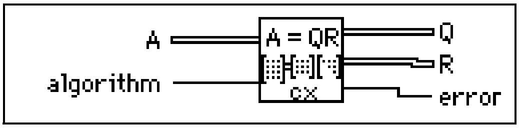


An SVD algorithm computes **PseudoInverse Matrix** A^+, and treats any singular values less than the **tolerance** as zeros. For a definition of the PseudoInverse of a matrix, see the *Solving Linear Equations and Matrix Inverses* section at the beginning of this chapter.

If Input matrix A is square and not singular, A^+ is the same as A^{-1}, but using the Complex Inverse Matrix VI to compute A^{-1} is more efficient than using this VI.

Complex QR Factorization

Performs QR factorization for a complex matrix A.

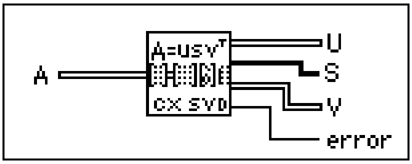


QR factorization is also called orthogonal-triangular factorization. It factors a complex matrix **A** into two matrices; one is an orthogonal matrix **Q**, the other is an upper triangular matrix **R**, so that *A* = *QR*. This VI provides three methods for the factorization: Householder, Givens, and Fast Givens.

You can use QR factorization to solve linear systems that contain less or more equations than unknowns.

Complex SVD Factorization

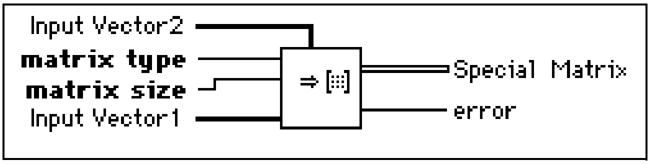
Performs the singular value decomposition (SVD) of a given *m*-by-*n*, complex matrix A with *m*>*n*.



SVD produces three matrices **U**, **S**, and **V**, so that $A = US_0V^H$, where *U* and *V* are orthogonal matrices, S_0 is an *n*-by-*n* diagonal matrix with the elements of array **S** on the diagonal in decreasing order. The diagonal elements are the singular values of A.

Create Special Complex Matrix

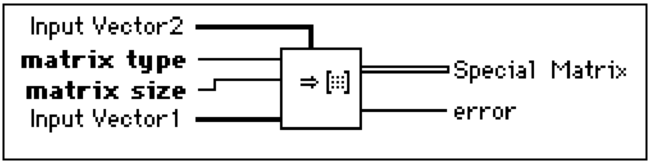
Generates a special, complex matrix based on the **matrix type**.



Let *n* represent **matrix size**, *X* represent **Input Vector1**, *nx* represent the size of *X*, and *Y* represent **Input Vector2**, *ny* represent the size of *Y*, and *B* represent the output **Special Matrix**.

Create Special Matrix

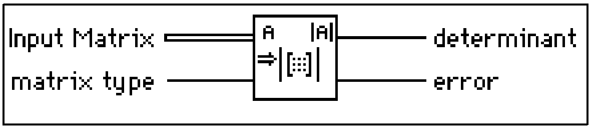
Generates a real, special matrix based on the **matrix type**.



Let *n* represent **matrix size**, *X* represent **Input Vector1**, *nx* represent the size of *X*, and *Y* represent **Input Vector2**, *ny* represent the size of *Y*, and *B* represent the output **Special Matrix**.

Determinant

Computes the **determinant** of a real, square matrix **Input Matrix**.



Let A be a square matrix that represents the **Input Matrix,** and let L and U represent the lower and upper triangular matrices, respectively, of A such that

$A = LU$,

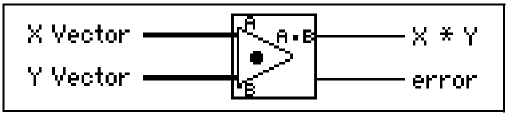
where the main diagonal elements of the lower triangular matrix L are arbitrarily set to one. The VI finds the **determinant** of A by the product of the main diagonal elements of the upper triangular matrix U

$$|A| = \prod_{i=0}^{n-1} u_{ii} ,$$

where $|A|$ is the **determinant** of **X**, and n is the dimension of **X**.

Dot Product

Computes the dot product of **X Vector** and **Y Vector**.



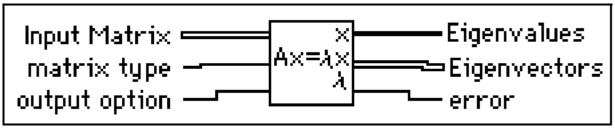
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains the dot product **X*Y** using the formula:

$$X^*Y = \sum_{i=0}^{n-1} x_i y_i \ ,$$

where n is the number of data points. Notice that the output value **X*Y** is a scalar value.

EigenValues & Vectors

Finds the eigenvalues and eigenvectors right of a square, real **Input Matrix**.



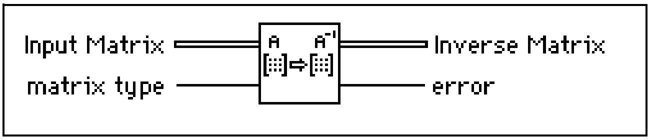
The eigenvalue problem is to determine the nontrivial solutions to the equation:

$AX = \lambda X$

where A is a n-by-n **Input Matrix**, X is a vector with n elements, and λ is a scalar. The n values of λ that satisfy the equation are the **Eigenvalues** of A and the corresponding values of X are the right **Eigenvectors** of A. A symmetric, real matrix always has real eigenvalues and eigenvectors.

Inverse Matrix

Finds the **Inverse Matrix** of the **Input Matrix**.



Let **A** be the **Input Matrix** and **I** be the identity matrix. You obtain the **Inverse Matrix** value by solving the system $AB = I$ for B.

If **A** is a nonsingular matrix, you can show that the solution to the preceding system is unique and that it corresponds to the **Inverse Matrix** of **A**:

$B = A^{-1}$,

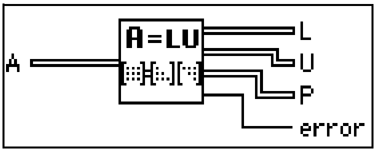
and **B** is therefore an **Inverse Matrix**. A nonsingular matrix is a matrix in which no row or column contains a linear combination of any other row or column, respectively.

Note: ***The numerical implementation of the matrix inversion is not only numerically intensive but, because of its recursive nature, is also highly sensitive to round-off errors introduced by the floating-point numeric coprocessor. Although the computations use the maximum possible accuracy, the VI cannot always solve for the system.***

You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.

LU Factorization

Performs the LU factorization of a real, square matrix **A.**

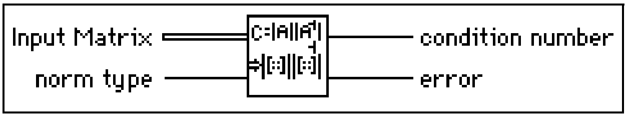


LU factorization factors the square matrix **A** into two triangular matrices; one is a lower triangular matrix **L** with ones on the diagonal, and the other is an upper triangular matrix **U**, so that $PA = LU$, where **P** is a permutation matrix, which serves as the identity matrix with some rows exchanged.

Factorization serves as a key step for inverting a matrix, computing the determinant of a matrix, and solving a linear equation.

Matrix Condition Number

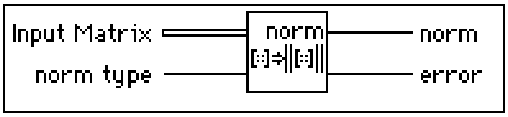
Computes the **condition number** of a real matrix **Input Matrix**.



The **condition number** of a matrix measures the sensitivity of a system solution of linear equations to errors in the data. It gives an indication of the accuracy of the results from a matrix inversion and a linear equation solution.

Matrix Norm

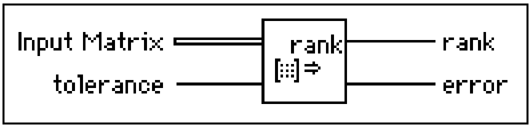
Computes the **norm** of a real matrix **Input Matrix**.



The norm of a matrix is a scalar that gives some measure of the magnitude of the elements in the matrix. Let **A** represent the **Input Matrix**, the norm of **A** is represented by $\|A\|_p$, where p can be 1,2,F,∞. Different values of p mean different types of norms that are computed.

Matrix Rank

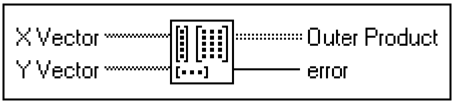
Computes the **rank** of a rectangular, real matrix **Input Matrix**.



Matrix rank is the number of singular values in the **Input Matrix** that are larger than the **tolerance**. **rank** is the maximum number of independent rows or columns in the **Input Matrix**.

Outer Product

Computes the outer product of **X Vector** and **Y Vector**.



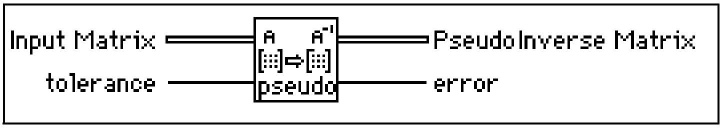
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains **Outer Product** using the formula:

$$a_{ij} = x_i\, y_j, \text{ for } \begin{cases} i = 0, 1, 2, ..., n-1 \\ j = 0, 1, 2, ..., m-1 \end{cases},$$

where A represents the 2D output sequence **Outer Product**, n is the number of elements in the input sequence **X Vector**, and m is the number of elements in the input sequence **Y Vector**.

PseudoInverse Matrix

Finds the **PseudoInverse Matrix** of a rectangular, real matrix **Input Matrix**.



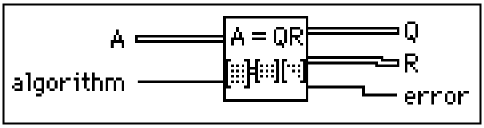
You compute **PseudoInverse Matrix** A^+ by using the SVD algorithm and any singular value less than the **tolerance,** which are set to zero. For a definition of the PseudoInverse of a matrix, see the *Solving Linear Equations and Matrix Inverses* section at the beginning of this chapter.

If Input matrix A is square and not singular, A^+ is the same as A^{-1}, but using the Inverse Matrix VI to compute A^{-1} is more efficient than using this VI.

☞ **Note:** ***This VI is not available with Base packages of LabVIEW.***

QR Factorization

Performs the QR factorization of a real matrix **A**.



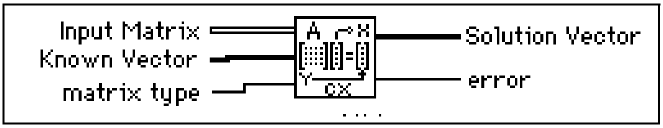
QR factorization is also called orthogonal-triangular factorization. It factors a real matrix **A** into two matrices. One is an orthogonal matrix **Q**, and the other is an upper triangular matrix **R**, so that $A = QR$. This VI provides three methods for the factorization: householder, givens, and fast givens.

You can use QR factorization to solve linear systems with more equations than unknowns.

☞ **Note:** ***This VI is not available with Base packages of LabVIEW.***

Solve Complex Linear Equations

Solves a complex, linear system $AX=Y$.



Let A represent the m-by-n **Input Matrix**, Y represent the set of m elements in the **Known Vector**, and X represent the set of n elements in the **Solution Vector** that solves for the system

$AX = Y$.

When $m > n$, the system has more equations than unknowns, so it is an overdetermined system. Since the solution that satisfies $AX=Y$ may not exist, the VI finds the least square solution X, which minimizes $\|AX-Y\|$.

When $m<n$, the system has more unknowns than equations, so it is an underdetermined system. It might have infinite solutions that satisfy $AX=Y$. The VI then selects one of these solutions.

When $m=n$, if A is a nonsingular matrix—no row or column is a linear combination of any other row or column, respectively—then you can solve the system for X by decomposing the **Input Matrix** A into its lower and upper triangular matrices, L and U, such that

$AX = LZ = Y$,

and

$Z = UX$

can be an alternate representation of the original system. Notice that Z is also an n element vector.

Triangular systems are easy to solve using recursive techniques. Consequently, when you obtain the L and U matrices from A, you can find Z from the $LZ = Y$ system and X from the $UX = Z$ system.

When $m \neq n$, A can be decomposed to an orthogonal matrix Q, and an upper triangular matrix R, so that $A=QR$, and the linear system can be represented by $QRX=Y$. You can then solve $RX=Q^{H}Y$.

You can easily solve this triangular system to get X using recursive techniques.

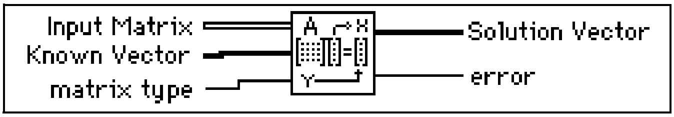
☞ **Note:** ***You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.***

The numerical implementation of the matrix inversion is numerically intensive and, because of its recursive nature, is also highly sensitive to round-off error introduced by the floating-point numeric coprocessor. Although the computations use the maximum possible accuracy, the VI cannot always solve for the system.

Note: ***This VI is not available with Base packages of LabVIEW.***

Solve Linear Equations

Solves a real linear system *AX*=*Y*.



Let *A* be an *m*-by-*n* matrix that represents the **Input Matrix**, *Y* be the set of *m* coefficients in **Known Vector**, and *X* be the set of *n* elements in **Solution Vector** that solves the system

$AX = Y.$

When *m*>*n*, the system has more equations than unknowns, so it is an overdetermined system. The solution that satisfies AX=Y may not exist, so the VI finds the least square solution X, which minimizes $\|AX - Y\|$.

When *m*<*n*, the system has more unknowns than equations, so it is an underdetermined systems. It may have infinite solutions that satisfy AX=Y. The VI finds one of these solutions.

In the case of *m*=*n*, if *A* is a nonsingular matrix–no row or column is a linear combination of any other row or column, respectively–then you can solve the system for *X* by decomposing the input matrix *A* into its lower and upper triangular matrices, *L* and *U*, such that

$AX = LZ = Y,$

and

$Z = UX$

can be an alternate representation of the original system. Notice that *Z* is also an *n* element vector.

Triangular systems are easy to solve using recursive techniques. Consequently, when you obtain the *L* and *U* matrices from *A*, you can find *Z* from the *LZ* = *Y* system and *X* from the *UX* = *Z* system.

In the case of $m \neq n$, A can be decomposed to an orthogonal matrix Q and an upper triangular matrix R, so that $A=QR$. The linear system can then be represented by $QRX=Y$. You can then solve $RX=Q^TY$.

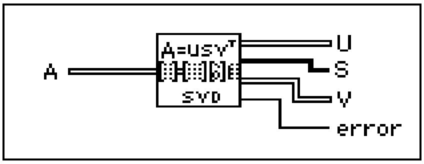
You can easily solve this triangular system to get x using recursive techniques.

☞ **Note:** ***You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.***

The numerical implementation of the matrix inversion is numerically intensive and, because of its recursive nature, is also highly sensitive to round-off error introduced by the floating-point numeric coprocessor. Although the computations use the maximum possible accuracy, the VI cannot always solve the system.

SVD Factorization

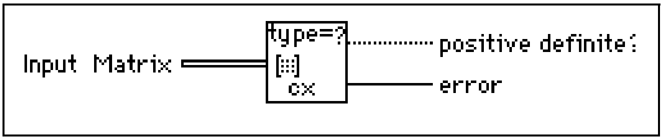
Performs the singular value decomposition (SVD) of a given *m*-by-*n* real matrix **A**, with *m*>*n*.



SVD produces three matrices U,S_0, and V so that A = US_0V^T, where U and V^T are orthogonal matrices, S_0 is an *n*-by-*n* diagonal matrix with the elements of array **S** on the diagonal in decreasing order.

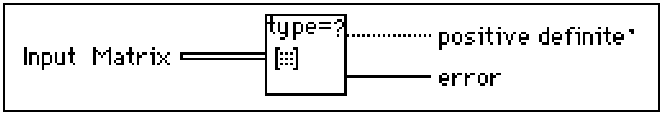
Test Complex Positive Definite

Tests whether the **Input Matrix** is a Positive Definite matrix.



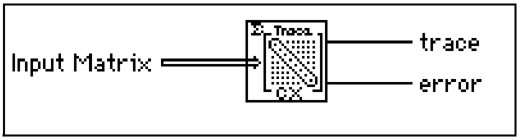
Test Positive Definite

Tests whether the **Input Matrix** is a Positive Definite matrix.



Trace

Finds the **trace** of **Input Matrix**.



Let *A* be a square matrix that represents **Input Matrix** and tr(*A*) be **trace**. The **trace** of A is the sum of the main diagonal elements of *A*

$$\mathrm{tr}(A) = \sum_{i=0}^{n-1} a_{ii} ,$$

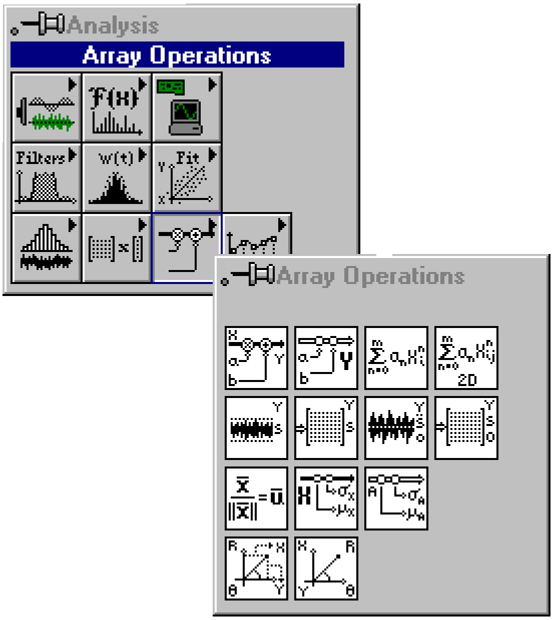
where *n* is the dimension of **Input Matrix**.

Chapter 47

Analysis Array Operation VIs

This chapter describes the VIs that perform common, one- and two-dimensional numerical array operations.

The following illustration shows the **Array Operations** palette, which you access by selecting **Functions»Analysis»Array Operations**.

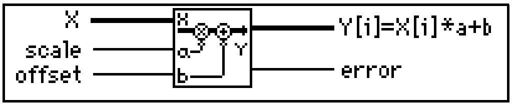


Array Operation VI Descriptions

The following Array Operation VIs are available.

1D Linear Evaluation

Performs a linear evaluation of the input array **X**.



The output array **Y[i] = X[i]*a + b** is given by

$$Y = aX + b ,$$

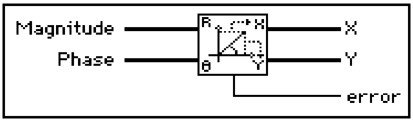
where *a* is the multiplicative **scale** constant, and *b* is the additive constant **offset**.

1D Polar To Rectangular

Converts two arrays of polar coordinates into two arrays of rectangular coordinates, according to the following formulas:

x = **magnitude** cos(**phase**)

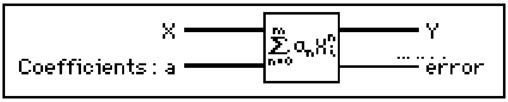
y = **magnitude** sin(**phase**).



Note: ***This VI is not available with Base packages of LabVIEW.***

1D Polynomial Evaluation

Performs a polynomial evaluation of **X** using **Coefficients: a**.



The output array **Y** is given by

$$Y = \sum_{n=0}^{m} a_n X^n ,$$

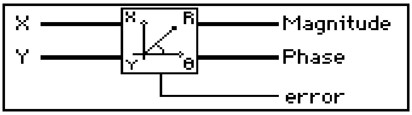
where *m* denotes the polynomial order.

1D Rectangular To Polar

Converts two arrays of rectangular coordinates into two arrays of polar coordinates, according to the following formulas:

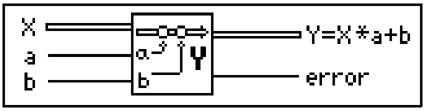
$$\text{magnitude} = \sqrt{x^2 + y^2}$$

$$\text{phase} = \tan^{-1}\left(\frac{y}{x}\right).$$



2D Linear Evaluation

Performs a linear evaluation of the two-dimensional input array **X**.



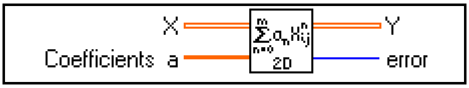
The two-dimensional output array **Y = X*a + b** is given by

$$Y = Xa + b,$$

where *a* denotes the multiplicative constant, and *b* denotes the additive constant.

2D Polynomial Evaluation

Performs a polynomial evaluation of the two-dimensional input array **X** using **Coefficients a**.



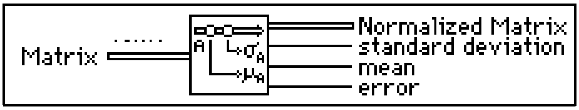
The two-dimensional output array **Y** is given by

$$Y = \sum_{n=0}^{m} a_n X^n,$$

where *m* denotes the polynomial order.

Normalize Matrix

Normalizes the 2D input **Matrix** using its statistical profile (μ, σ), where μ is the **mean** and σ is the **standard deviation**, to obtain a **Normalized Matrix** whose statistical profile is (0,1).



The VI obtains **Normalized Matrix** using

$$B = \frac{A - \mu}{\sigma},$$

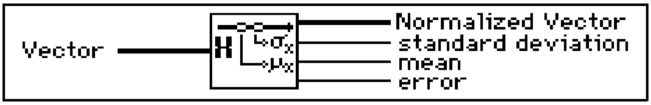
$$\mu = \frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_{ij}}{n \bullet m},$$

$$\sigma = \sqrt{\left| \frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (a_{ij} - \mu)^2}{n \bullet m} \right|},$$

where *B* represents the 2D output sequence **Normalized Matrix**, *A* represents the 2D input sequence **Matrix** with *n* rows and *m* columns, and a_{ij} is the element of *A* on the i^{th} row and j^{th} column.

Normalize Vector

Normalizes the input **Vector** using its statistical profile (μ,σ), where μ is the **mean** and σ is the **standard deviation**, to obtain a **Normalized Vector** whose statistical profile is (0,1).



The VI obtains **Normalized Vector** using

$$Y = \frac{X - \mu}{\sigma},$$

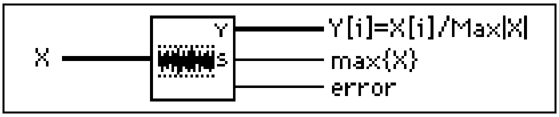
$$\mu = \frac{\sum_{i=0} x_i}{n},$$

$$\sigma = \sqrt{\left|\frac{\sum_{i=0}^{n-1} (x_i - \mu)^2}{n}\right|},$$

where *Y* represents the output sequence **Normalized Vector**, and *X* represents the input sequence **Vector** of length *n*, and x_i is the i^{th} element of *X*.

Quick Scale 1D

Determines the maximum absolute value of the input array **X** and then scales **X** using this value.



The output array **Y[i] = X[i]/Max|X|** is given by

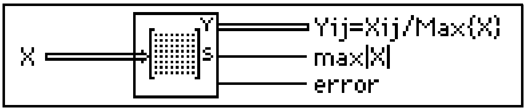
$$Y = \frac{X}{s},$$

where *s* is the maximum absolute value in **X**.

You can use this VI to normalize sequences within the range [–1:1]. This VI is particularly useful if the sequence is a zero mean sequence.

Quick Scale 2D

Determines the maximum absolute value of the input array **X** and then scales **X** using this value.



The output array **Yij = Xij/Max{X}** is given by

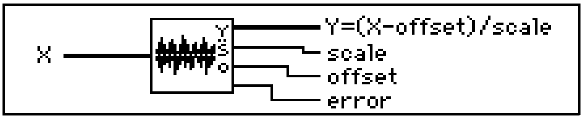
$$Y = \frac{X}{s},$$

where *s* denotes the maximum absolute value in **X**.

You can use this VI to normalize sequences within the range [–1:1]. This VI is particularly useful if the sequence is a zero mean sequence.

Scale 1D

Determines **scale** and **offset** and then scales the input array **X** using these values.



The output array **Y** is given by

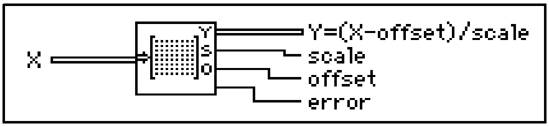
$$Y = \frac{X - offset}{scale},$$

scale = 0.5(*max* – *min*), and **offset** = *min* + **scale**, where *max* denotes the maximum value in **X**, and *min* denotes the minimum value in **X**.

You can use this VI to normalize any numerical sequence with the assurance that the range of the output sequence is [–1:1].

Scale 2D

Determines **scale** and **offset** and then scales **X** using these values.



The two-dimensional output array **Y = (X – offset)/scale** is given by

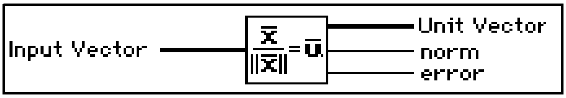
$$Y = \frac{X - \text{offset}}{scale},$$

scale = 0.5(*max* – *min*), and **offset** = *min* + 0.5 **scale**, where *max* denotes the maximum value in **X**, and *min* denotes the minimum value in **X**.

You can use this VI to normalize any numerical sequence with the assurance that the range of the output sequence is [–1:1].

Unit Vector

Finds the **norm** of the **Input Vector** and obtains its corresponding **Unit Vector** by normalizing the original **Input Vector** with its **norm**.



Let *X* represent the input **Input Vector**; **norm** is given by

$$\|X\| = \sqrt{x_0^2 + x_1^2 + \ldots + x_{n-1}^2},$$

where $\|X\|$ is **norm**, and the VI calculates **Unit Vector,** *U*, using

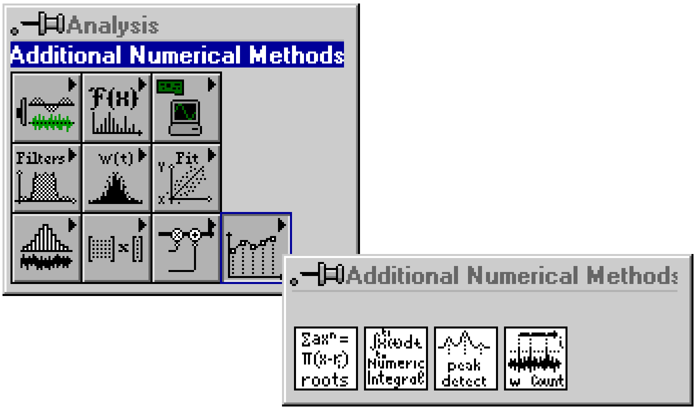
$$U = \frac{X}{\|X\|}.$$

Chapter 48

Analysis Additional Numerical Method VIs

This chapter describes the VIs that use numerical methods to perform root-finding, numerical integration, and peak detection.

The following illustration shows the **Additional Numerical Methods** palette, which you access by selecting **Functions»Analysis»Additional Numerical Methods**.

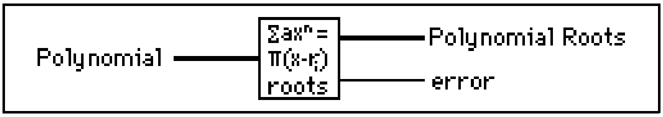


Additional Numerical Method VI Descriptions

The following Additional Numerical Method VIs are available.

Complex Polynomial Roots

Finds the complex roots of a complex polynomial.

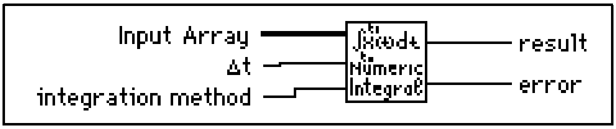


This VI uses a modified, complex Newton method to determine the *n* complex roots (some of which may be real, with a zero imaginary part), of the general complex polynomial:

$$a_0+a_1x+a_2x^2+...+a_{n-1}x^{n-1}+a_nx^n.$$

Numeric Integration

Performs a numeric integration on the input array of data using one of four, popular numeric integration methods.



☞ **Note:** ***If the number of points provided for a certain chosen method does not contain an integral number of partial sums, then the method is applied for all possible points. For the remaining points, the next possible lower order method is used. For example, if the Bode method is selected, the following table shows what this VI evaluates for different numbers of points:***

Number of Points	Partial Evaluations Performed
224	56 Bode
225	56 Bode, 1 Trapezoidal
226	56 Bode, 1 Simpsons’
227	56 Bode, 1 Simpsons’ 3/8
228	57 Bode

So, if 227 points were provided and the Bode Method was chosen, the VI would arrive at the result by performing 56 Bode Method partial evaluations and one Simpsons’ 3/8 Method evaluation.

Each of the methods depend on the sampling interval (**dt**) and compute the integral using successive applications of a basic formula in order to perform partial evaluations, which depend on some number of adjacent points. The number of points used in each partial evaluation represents the order of the method. The result is the summation of these successive partial evaluations.

$$\text{result} = \left(\int_{t0}^{t1} f(t)dt \approx \sum_{j} \text{partial sums} \right),$$

where *j* is a range dependent on the number of points and the method of integration.

The basic formulas for the computation of the partial sum of each rule in ascending method order are:

Trapezoidal: ($x[i] + x[i+1]$)*dt, $k = 1$

Simpsons': ($x[2i] + 4x[2i+1] + x[2i+2]$)*dt/3, $k = 2$

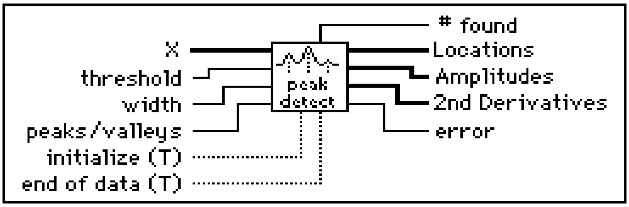
Simpsons' 3/8: ($3x[3i] + 9x[3i+1] + 9x[3i+2] + 3x[3i+3]$) * dt/8, $k = 3$

Bode: ($14x[4i] + 64x[4i+1] + 24x[4i+2] + 64x[4i+3] + 14x[4i+4]$)*dt/45, $k = 4$
for $i = 0, k, 2k, 3k, 4k...$, Integral Part of $[(N-1)/k]$

where *N* is the number of data points, *k* is an integer dependent on the method, and *x* is the input array.

Peak Detector

Finds the location, amplitude, and second derivative of peaks or valleys in the input array.



The data set can be passed to the VI as a single array or as consecutive blocks of data.

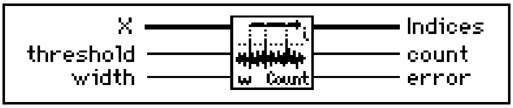
This VI is based on an algorithm that fits a quadratic polynomial to sequential groups of data points. The number of data points used in the fit is specified by **width**.

For each peak or valley, the quadratic fit is tested against the threshold level: peaks with heights lower than the threshold or valleys with troughs higher than the threshold are ignored. **peaks/valleys** are detected only after approximately **width**/2 data points have been processed beyond **peaks/valleys** locations. This delay has implications only for real time processing.

The VI must be notified when the first and last blocks are passed into the VI, so that the VI can initialize and then release data internal to the peak detection algorithm.

Threshold Peak Detector

Analyzes the input sequence **X** for valid peaks and keeps a **count** of the number of peaks encountered and a record of **Indices**, which locates the points that exceed the **threshold** in a valid peak. A peak is valid where the elements of **X** exceed the **threshold** and then return to a value less than or equal to the **threshold**, and the number of elements that exceed the **threshold** is at least equal to **width**.



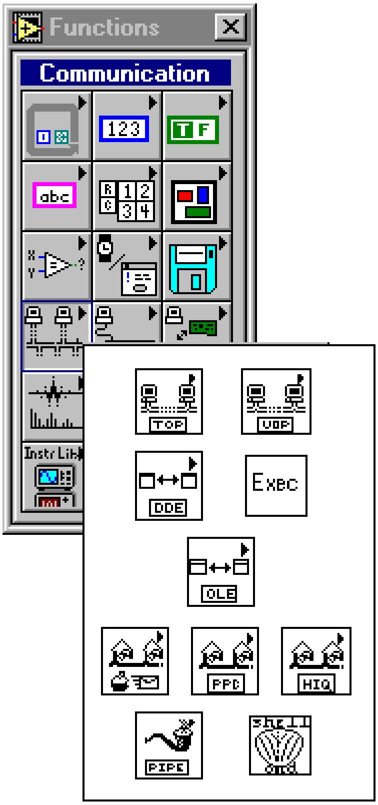
Chapter 49

Introduction to LabVIEW Communication VIs and Functions

This chapter introduces the way LabVIEW handles networking and interapplication communications, and introduces the Communication functions and VIs, descriptions of which comprise Chapter 50 to Chapter 55.

You can find the Communication VIs in the **Functions** palette from your block diagram in LabVIEW. The Communication VIs are located near the middle of the **Functions** palette.

To access the **Communications** palette, select **Functions»Communications**, as shown in the following illustration.



The **Communications** palette consists of the following subpalettes:

- TCP
- UDP
- DDE **(Windows only)**
- OLE **(Windows only)**
- HiQ

If you have LabVIEW for the Macintosh, the following additional subpalettes are available:

- Apple Event
- Program to Program Communications

If you have a computer running a UNIX operating system and LabVIEW, the following additional subpalette is available:

- Named Pipes

LabVIEW for Windows and LabVIEW for UNIX also include the System Exec VI.

You can find information about these LabVIEW features online by using the LabVIEW Help window (**Help»Show Help**). When you place the cursor on a VI icon, the wiring diagram and parameter names for that VI appear in the Help window. You can also find information for front panel controls or indicators by placing the cursor over the control or indicator with the Help window open.

In addition to the Help window, LabVIEW has more extensive online information available. To access this information, select **Help»Online Reference**. For most block diagram objects, you can select **Online Reference** from the object's pop-up menu to access the online description.

LabVIEW Communication VIs and Functions Overview

For the purpose of this discussion, *networking* refers to communication between multiple processes. The processes can optionally run on separate computers. This communication usually occurs over a hardware network, such as ethernet or LocalTalk.

One main use for networking in software applications is to allow one or more applications to use the services of another application. For example, the application providing services (the server) could be either a data collection application running on a dedicated computer, or a database program providing information for other applications.

The purpose of this discussion is to introduce you to the terminology used in networking and communication applications, and to give you an overview of how to program networked applications.

Introduction to Communication Protocols

For communication between processes to work, the processes must use a common communications language, referred to as a *protocol*.

A communication protocol lets you specify the data that you want to send or receive and the location of the destination or source, without having to worry about how the data gets there. The protocol translates your commands into data that network drivers can accept. The network drivers then take care of transferring data across the network as appropriate.

Several networking protocols have emerged as accepted standards for communications. In general, one protocol is not compatible with a different protocol. Thus, in communication applications, one of the first things you must do is decide which protocol to use. If you want to communicate with an existing, off the shelf application, then you have to work within the protocols supported by that application.

When you are actually writing the application, you have more flexibility in choosing a protocol. Factors that affect your protocol choice include the type of machines the processes will run on, the kind of hardware network you have available, and the complexity of the communication that your application will need.

Several protocols are built into LabVIEW, some of which are specific to a type of computer. LabVIEW uses the following protocols to communicate between computers:

- **TCP**—Available on all computers
- **UDP**—Available on all computers
- **DDE**—Available on the PC, for communication between Windows applications
- **OLE**—Available for use with Windows 95 and Windows NT
- **AppleEvents**—Available on the Macintosh, for sending messages between Macintosh applications
- **PPC**—Available on the Macintosh, for sending and receiving data between Macintosh applications

Each protocol is different, especially in the way they refer to the network location of a remote application. They are incompatible with each other, so if you want to communicate between a Macintosh and a PC, you must use a protocol compatible with both, such as TCP.

Other communication options provided by LabVIEW include:

- **System Exec VI**, which allows you to execute a system level command. There are actually two System Exec VIs, one for use with all versions of Windows, the other with Sun and HP-UX.

- **Named Pipes**, available on UNIX only
- HiQ®, available on the Macintosh and PC only

File Sharing vs Communication Protocols

Before you get too deeply involved in communication protocols, consider whether another approach is more appropriate for your application. For instance, consider an application where a dedicated system acquires data and you want the data recorded on a different computer.

You could write an application that uses networking protocols to send data from the acquisition computer to the data repository machine, where a separate application collects the data and stores it on disk.

A simpler method is to use the filesharing capabilities available on most networked computers. With filesharing, drivers that are part of the operating system let you connect to other machines. The remote machine's disk storage is treated as an extension of your own disk storage. Once you connect two systems, filesharing usually makes this connection transparent, so that any application can write to the remote disk as if connected locally.

Filesharing is frequently the simplest method for transferring data between machines.

Client/Server Model

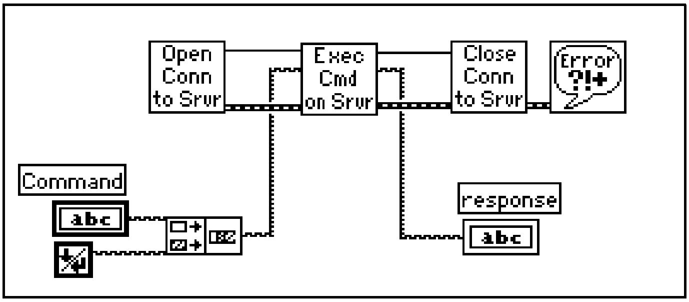
The client/server model is a common model for networked applications. In the client/server model, one set of processes (clients) request services from another set of processes (servers).

For example, in your application you could set up a dedicated computer for acquiring measurements from the real world. The computer acts as a server when it provides data to other computers on request. It acts as a client when it requests another application, such as a database program, to record the data that it acquires.

In LabVIEW, you can use client and server applications with all protocols except Macintosh AppleEvents. You can use AppleEvents to send commands to other applications. You cannot set up a command server in LabVIEW using AppleEvents. If you need server capabilities on the Macintosh, use either TCP, UDP or PPC.

A General Model for a Client

The following block diagram shows what a simplified model for a client looks like in LabVIEW.



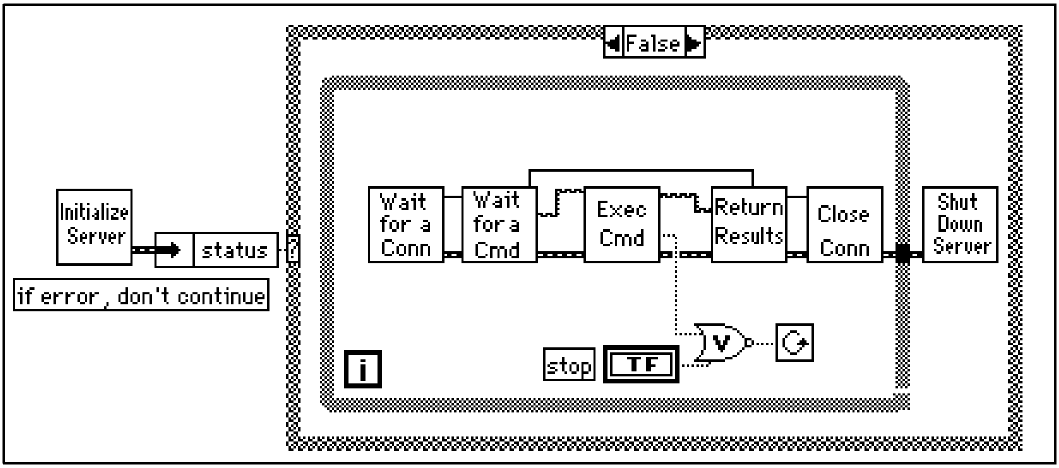
In the preceding diagram, LabVIEW first opens a connection to a server. It then sends a command to the server, gets a response back, and closes the connection to the server. Finally, it reports any errors that occurred during the communication process.

For higher performance, you can process multiple commands once the connection is open. After the commands are executed, you can close the connection.

This basic block diagram structure serves as a model and is used elsewhere in this manual to demonstrate how to implement a given protocol in LabVIEW.

A General Model for a Server

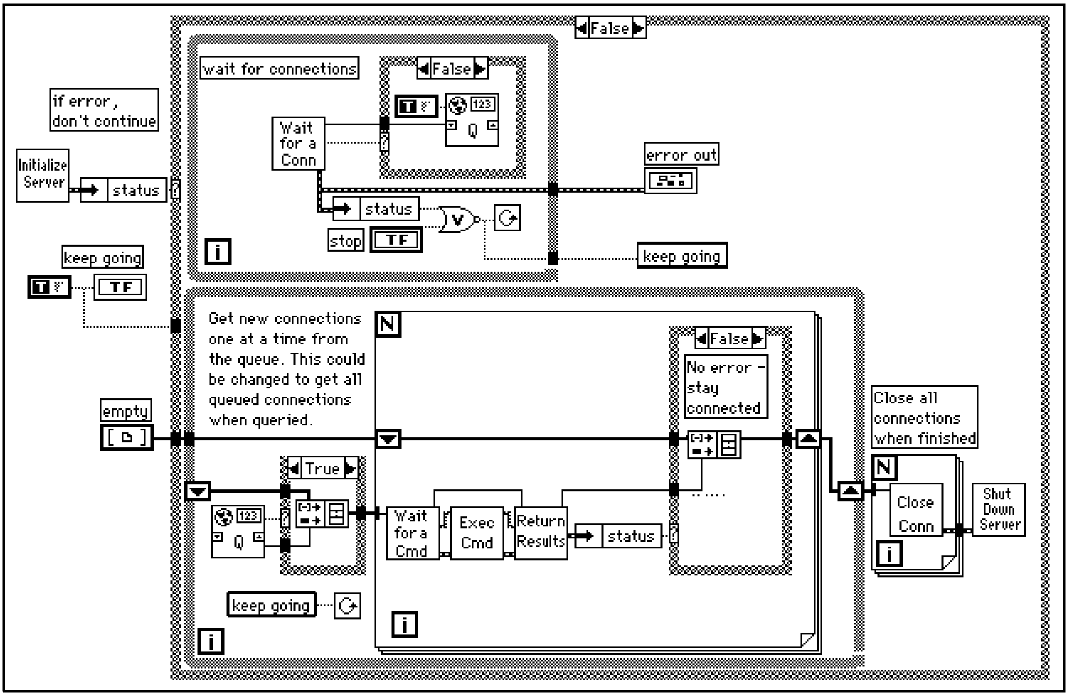
The following block diagram shows a simplified model for a server in LabVIEW.



In the preceding diagram, LabVIEW first initializes the server. If the initialization is successful, LabVIEW goes into a loop, where it waits for a connection. Once the connection is made, LabVIEW waits to receive a command. LabVIEW executes the command and returns the results. The connection is then closed. LabVIEW repeats this entire process until it is shut down locally by pressing a stop button on the front panel, or remotely by sending a command to shut the VI down.

This VI does not report errors. It may send back a response indicating that a command is invalid, but it does not display a dialog when an error occurs. Because a server might be unattended, consider carefully how the server should handle errors. You probably do not want a dialog box to be displayed, because that requires user interaction at the server (someone would have to press the OK button). However, you might want LabVIEW to write a log of transactions and errors to a file or a string.

You can increase performance by allowing the connection to stay open, so that you can receive multiple commands, but this blocks others clients from connecting until the current client disconnects. If the protocol supports multiple simultaneous connections, you can restructure LabVIEW to handle multiple clients simultaneously, as shown in the following diagram.



The preceding diagram uses LabVIEW's multitasking capabilities to run two loops simultaneously. One loop continuously waits for a connection. When a connection is received, it is added to a queue. The other loop checks each of the open connections and executes any commands that have been received. If an error occurs on one of the connections, the connection is disconnected. When the user aborts the server, all open connections are closed. This basic block diagram structure is a model which is used elsewhere in this manual to demonstrate how to implement a given protocol in LabVIEW.

TCP/IP (all platforms)

TCP/IP is a suite of communication protocols, originally developed for the Defense Advanced Research Projects Agency (DARPA). Since its development, it has become widely accepted, and is available on a number of computer systems.

The name TCP/IP comes from two of the best known protocols of the suite, the Transmission Control Protocol (TCP) and the Internet Protocol (IP). TCP, IP, and the User Datagram Protocol (UDP) are the basic tools for network communication.

TCP/IP enables communication over single networks or multiple interconnected networks, which are known as an internetwork or internet. The individual networks can be separated by great geographical distances. TCP/IP routes data from one network or internet computer to another. Because TCP/IP is available on most computers, it can transfer information between diverse systems.

Internet Protocol (IP) transmits data across the network. This low level protocol takes data of a limited size and sends it as a *datagram* across the network. IP is rarely used directly by applications, because it does not guarantee that the data will arrive at the other end. Also, when you send several datagrams they sometimes arrive out of order, or are delivered multiple times, depending on how the network transfer occurs. UDP, which is built on top of IP, has similar problems.

TCP is a higher level protocol that uses IP to transfer data. TCP breaks data into components that IP can manage. It also provides error detection and ensures that data arrives in order without duplication. For these reasons, TCP is usually the best choice for network applications.

Internet Addresses

Each host on an IP network has a unique 32-bit internet address. This address identifies the network on the internet to which the host is attached, and the specific computer on that network. You use this address to identify the sender or receiver of data. IP places the address in the datagram headers, so that each datagram is routed correctly.

One way of describing this 32-bit address is the IP dotted decimal notation. This divides the 32-bit address into four 8-bit numbers. The address is written as the four integers, separated by decimal points. For example, the 32-bit address

```
10000100     00001101     00000010    00011110
```

is written in dotted decimal notation as

```
132.13.2.30
```

Another way of using the 32-bit address is by names that are mapped to the IP address. Network drivers usually perform this mapping by consulting a local *hosts* file that contains name to address mappings, or consulting a larger database using the Domain Name System to query other computer systems for the address for a given name. Your network configuration dictates the exact mechanism for this process, which is known as *hostname resolution*.

Setup

Before you can use TCP/IP, you need to make sure that you have the right setup. This setup varies, depending on the computer you use.

Setup for Your System

UNIX

TCP/IP support is built-in. Assuming your network is configured properly, no additional setup for LabVIEW is necessary.

Macintosh

TCP/IP is built in to Macintosh operating system version 7.5 and later. To use TCP/IP with an earlier system, you need to install the MacTCP driver software, available from the Apple Programmer Developer Association (APDA). You can contact APDA at (800) 282-2732 for information on licensing the MacTCP driver. LabVIEW also works with Open Transport.

Windows 3.x

To use TCP/IP, you must install an ethernet board along with its low-level driver. In addition, you must purchase and install TCP/IP software that includes a Windows Sockets (WinSock) DLL conforming to standard 1.1. WinSock is a standard interface that enables application communication with a variety of network drivers. Several vendors provide network software that includes the WinSock DLL. Install the ethernet board, the board drivers, and the WinSock DLL according to the software vendor instructions.

Several vendors supply WinSock drivers that work with a number of boards. You can contact the vendor of your board to inquire if they offer a WinSock DLL you can use with the board. Install the WinSock DLL according to vendor instructions.

National Instruments has tested a number of WinSock DLLs to verify which work correctly. These tests showed that many DLLs do not fully comply with the standard, so you may want to try a demo version of a DLL before you buy the real version. You can usually obtain a demo version from the manufacturer. Most demo versions are fully functional, but they expire after a certain amount of time.

If you have access to the internet, several of these demos are available by anonymous ftp from `sunsite.unc.edu`. in the directory `/pub/micro/pc-stuff/ms-windows/winsock/packages`. Refer to your LabVIEW Release Notes for a detailed list of WinSock DLLs tested by National Instruments.

Windows 95 and Windows NT

TCP support is built-in to Windows NT. You do not need to use a third-party DLL to communicate using TCP.

LabVIEW and TCP/IP

You can use the TCP/IP suite of protocols with LabVIEW on all platforms. LabVIEW has a set of TCP and UDP VIs that you can use to create client or server VIs.

TCP versus UDP

If you are writing both the client and server, and your system can use TCP/IP, then TCP is probably the best protocol to use because it is a reliable, connection-based protocol. UDP is a connectionless protocol with higher performance, but it does not ensure reliable transmission of data.

TCP Client Example

The following discussion is a generalized description of how to use the components of the Client block diagram model with the TCP protocol.

Open
Conn
to Srvr

Use the TCP Open Connection VI to open a connection to a server. You must specify the internet address of the server, as well as the *port* for the server. The address identifies a computer on the network. The port is an additional number that identifies a communication channel on the computer that the server uses to listen for communication requests. When you create a TCP server, you specify the port that you want the server to use for communication.

Exec Cmd on Srvr

To execute a command on the server, use the TCP Write VI to send the command to the server. You then use the TCP Read VI to read back results from the server. With the TCP Read VI, you must specify the number of characters you want to read. This can be awkward, because the length of the response may vary. The server can have the same problem with the command, because the length of a command can vary.

The following are several methods you can use to address varying sized commands:

- Precede the command and the result with a fixed size parameter that specifies the size of the command or result. In this case, read the size parameter, and then read the number of characters specified by the size. This option is efficient and flexible.
- Make each command and result a fixed size. When a command is smaller than the size, you can pad it out to the fixed size.
- Follow each command and result with a specific terminating character. To read the data, you then need to read data in small chunks until you get the terminating character.

Close Conn to Srvr

Use the TCP Close Connection VI to close the connection to the server.

Timeouts and Errors

The preceding section discussed communication protocol for the server. When you design a network application consider carefully what should happen if something fails. For example, if the server crashes, how would each of the client VIs handle it?

One solution is to make sure that each VI has a timeout. This way, if something fails to produce results, after a certain amount of time, the client will continue execution. In continuing, the client can try to reestablish execution, or it can report the error, and if necessary, shut the client application down gracefully.

TCP Server Example

The following discussion explains how you can use TCP to fulfill each component of the general server model.

Initialize Server

No initialization is necessary with TCP, so this step can be left out.

Wait for a Conn

Use the TCP Listen VI to wait for a connection. You must specify the port that will be used for communication. This port must be the same port that the client will attempt to connect. For more information, see the *TCP Client Example* section in this chapter.

Wait for a Cmd

If a connection is established, read from that port to retrieve a command. As discussed in the TCP Client example, you must decide the format for commands. If commands are preceded by a length field, first read the length field, and then read the amount of data indicated by the length field.

Exec Cmd

Execution of a command should be protocol independent, because it is done on the local computer. When finished, pass the results to the next stage, where they are transmitted to the client.

Return Results

Use the TCP Write VI to return results. As discussed in the TCP Client example, the data must be in a form that the client can accept.

Close Conn

Use the TCP Close Connection VI to close the connection.

Shut Down Server

This step can be left out with TCP, because everything is finished after you close the connection.

TCP Server with Multiple Connections

TCP handles multiple connections easily. You can use the methods described in the preceding section to implement the components of a server with multiple connections.

DDE (Windows Only)

Dynamic Data Exchange (DDE) is a protocol for exchanging data between Windows applications.

In TCP/IP communications, applications open a line of communication and then transfer raw data. DDE works at a higher level, where applications send messages to each other to exchange information. One simple message is to send a command to another application. Most of the other messages deal with transferring data, where the data is referenced by name.

A DDE client initiates a conversation with another application (a DDE server) by sending a connect message. After establishing a connection, the client can send commands to the server and change or request the value of data that the server manages.

A client can request data from a server by a request or an advise. The client uses a request to ask for the current value of the data. If a client wants to monitor a value over a period of time, the client must request to be advised of changes. By asking to be advised of data value, the client establishes a link between the client and server through which the server notifies the client when the data changes. The client can stop monitoring the value of the data by telling the server to stop the advise link.

When the DDE communication for a conversation is complete, the client sends a close conversation message to the server.

DDE is most appropriate for communication with standard off the shelf applications such as Microsoft Excel.

With LabVIEW you can create VIs that act as clients to other applications (meaning they request or send data to other applications). You can also create VIs that act as servers that provide named information for access by other applications. As a server, LabVIEW does not use *connection*-based communication. Instead, you provide named information to other applications, which can then read or set the values of that information by name.

Services, Topics, and Data Items

With TCP/IP, you identify the process you want to talk to by its computer address and a port number. With DDE, you identify the application you want to talk to by referencing the name of a service and a topic. The server decides on arbitrary service and topic names. A given server generally uses its application name for the service, but not necessarily. That server can offer several topics that it is willing to communicate. With Excel, for example, the topic might be the name of a spreadsheet.

To communicate with a server, first find the names of the service and topic that you want to discuss. Then open a conversation using these two names to identify the server.

Unless you are going to send a command to the server, you usually work with data items that the server is willing to talk about. You can treat these as a list of *variables* that the server lets you manipulate. You can change variables by name, supplying a new value for the variable. Or, you can request the values of variables by name.

Examples of Client Communication with Excel

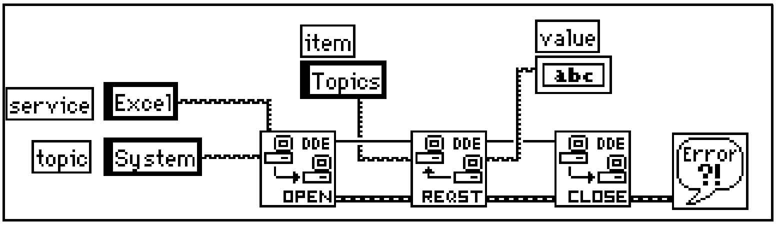
Each application that supports DDE has a different set of services, topics, and data items that it can talk about. For example, two different spreadsheet programs can take very different approaches to how they specify spreadsheet cells. To find out what a given application supports, consult the documentation that came with that application.

Microsoft Excel, a popular spreadsheet program for Windows, has DDE support. You can use DDE to send commands to Excel. You can also manipulate and read spreadsheet data by name. For more information on how to use DDE with Excel, refer to the *Microsoft Excel User's Guide 2*.

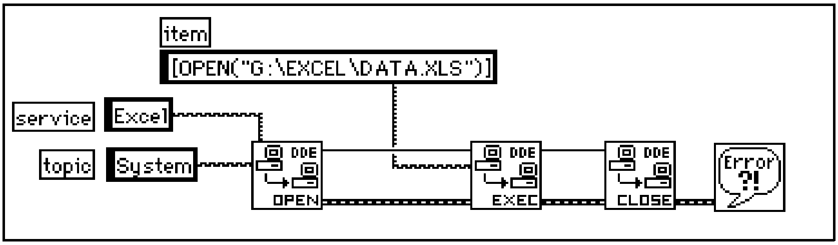
With Excel, the service name is *Excel*. For the topic, you use the name of an open document, such as spreadsheet document, or the word `System`.

If you use the name `System`, you can request information about the status of Excel, or send general commands to Excel (commands that are not directed to a specific spreadsheet). For instance, for the topic `System`, Excel will talk about items such as `Status`, which will have a value of `Busy` if Excel is busy, or `Ready` if Excel is ready to execute commands). Another, more useful data item you can use when the topic is `Status` is `Topics`, which returns a list of topics Excel will talk about, including all open spreadsheet documents and the `System` topic.

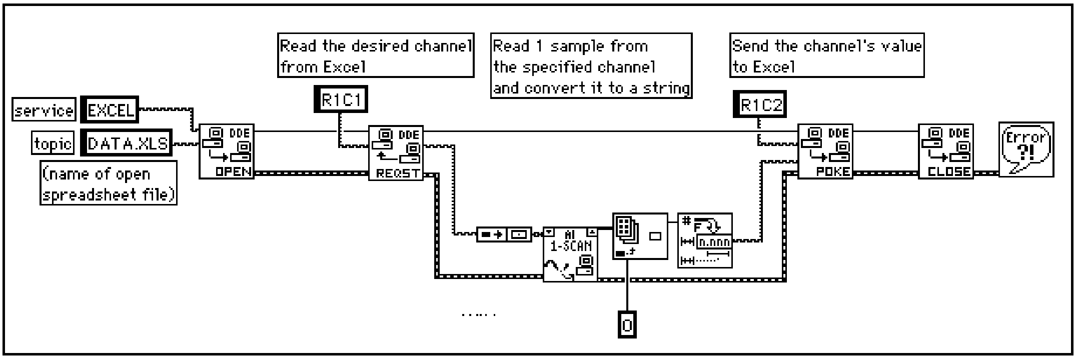
The following VI shows how you can use the Topics command in LabVIEW. The value returned is a string containing the names of the open spreadsheets and the work Excel.



Another way you can use the System topic with Excel is to instruct Excel to open a specific document. To do this, you use the DDE Execute.vi to send an Excel Macro to Excel that instructs Excel to open the document, as shown in the following LabVIEW diagram.



After you open a spreadsheet file, you can send commands to the spreadsheet to read cell values. In this case, your topic is the spreadsheet document name. The item is the name of a cell, a range of cells, or a named section of a spreadsheet. For example, in the following diagram LabVIEW can retrieve the value in the cell at row one column one. It then acquires a sample from the specified channel, and sends the resulting sample back to Excel.


Read the desired channel from Excel
Read 1 sample from the specified channel and convert it to a string
Send the channel's value to Excel
service
EXCEL
topic
DATA.XLS
(name of open spreadsheet file)
R1C1
R1C2
DDE
OPEN
DDE
REQST
AI
1-SCAN
DDE
POKE
DDE
CLOSE
Error
?!
0

LabVIEW VIs as DDE Servers

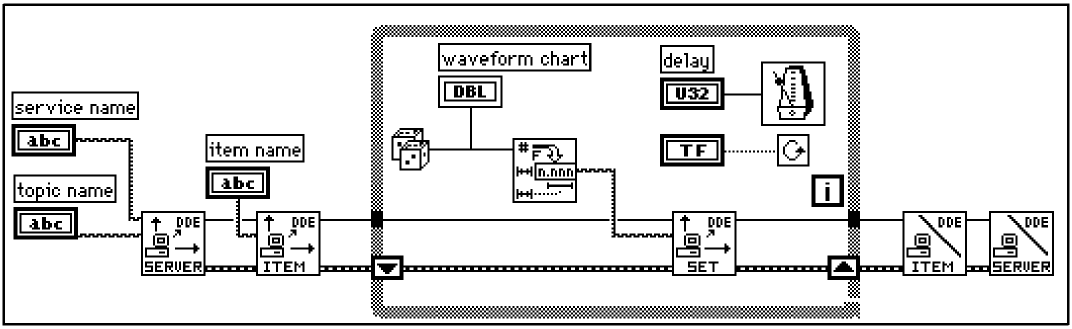
You can create LabVIEW VIs that act as servers for data items. The general concept is that a LabVIEW VI indicates that it is willing to provide information regarding a specific service in topic. LabVIEW can use any name for the service and topic name. It might specify the service name to be the name of the application (`LabVIEW`), and the topic name to be either the name of the Server VI, or a general classification for the data it provides, such as `Lab Data`.

The Server VI then registers data items for a given service that it will talk about. LabVIEW remembers the data names and their values, and handles communication with other applications regarding the data. When the server VI changes the value of data that is registered for DDE communication, LabVIEW notifies any client applications that have requested notification concerning that data. In the same way, if another application sends a `Poke` message to change the value of a data item, LabVIEW changes this value.

You cannot use the DDE Execute Command with a LabVIEW VI acting as a server. If you want to send a command to a VI, you must send the command using data items.

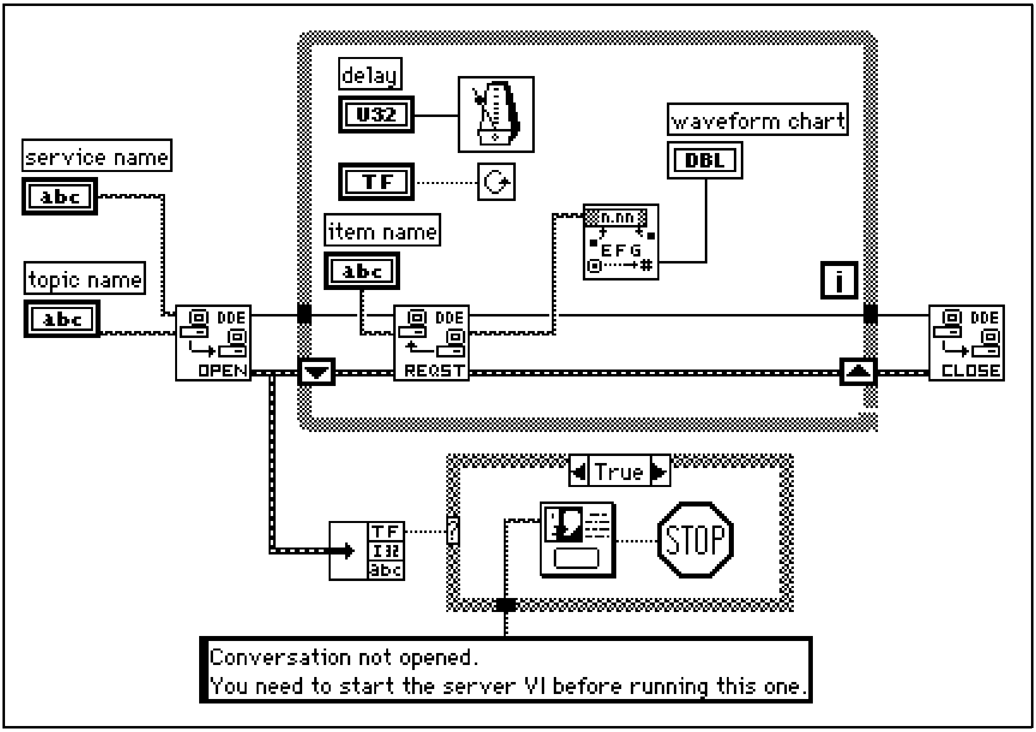
Also, notice that LabVIEW does not currently have anything like the System topic that Excel provides. The LabVIEW application is not itself a server to which you can send commands or request status information. It is important to understand that LabVIEW VIs act as servers and that at this time LabVIEW does not itself provide any services to other applications.

The following example shows how to create a DDE Server VI that provides data to other client applications. In this case, the data is a random number. You can easily replace the random number with real world data from data acquisition boards or devices connected to the computer by GPIB, VXI, or serial connections.



The VI in the preceding diagram registers a server with LabVIEW. The VI registers an item that it is willing to provide to clients. In the loop, the VI periodically sets the value of the item. As mentioned earlier, LabVIEW notifies other applications that data is available. When the loop is complete, the VI finishes by unregistering the item and unregistering the server.

The clients for this VI can be any applications that understand DDE, including other LabVIEW VIs. The following diagram illustrates a client to the VI shown in the previous diagram. It is important that the service, topic, and item names are the same as the ones used by the server.

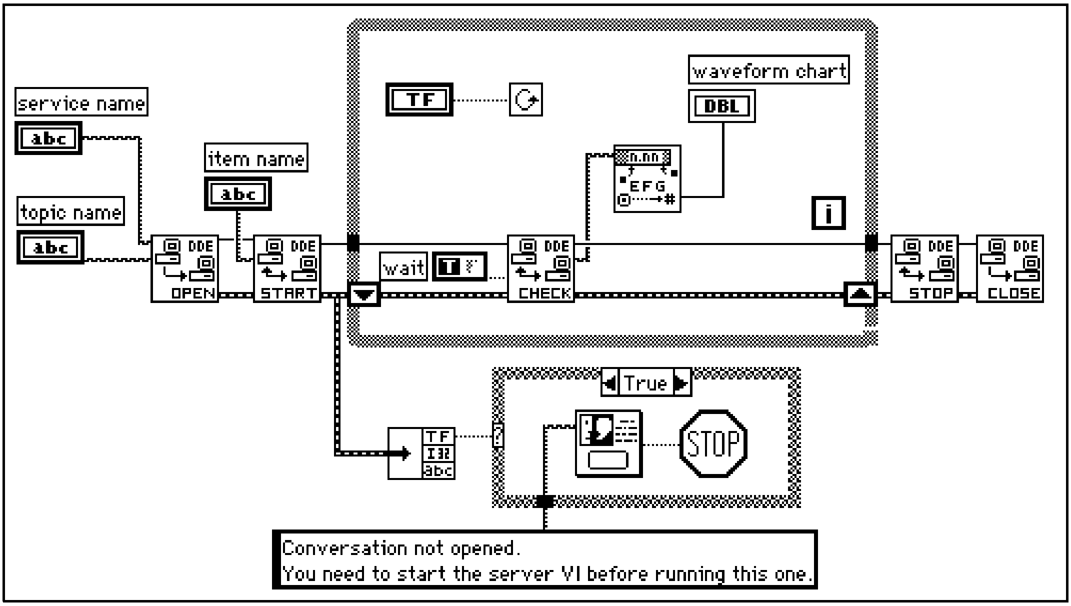


Conversation not opened.
You need to start the server VI before running this one.

Requesting Data versus Advising Data

The previous client example used the DDE Request VI in a loop to retrieve data. With DDE Request, the data is retrieved immediately, regardless of whether you have seen the data before. If the server and the client do not loop at exactly the same rate, you can duplicate or miss data.

One way to avoid duplicating data is to use the DDE Advise VIs to request notification of changes in the value of a data item. The following diagram shows how you can implement this scheme.



Conversation not opened.
You need to start the server VI before running this one.

In the preceding diagram, LabVIEW opens a conversation. It then uses the DDE Advise Start VI to request notification of changes in the value of a data item. Every time through the loop, LabVIEW calls the DDE Advise Check VI, which waits for a data item to change its value. When the loop is finished, LabVIEW ends the advise loop by calling the DDE Advise Stop VI, and closing the conversation.

Synchronization of Data

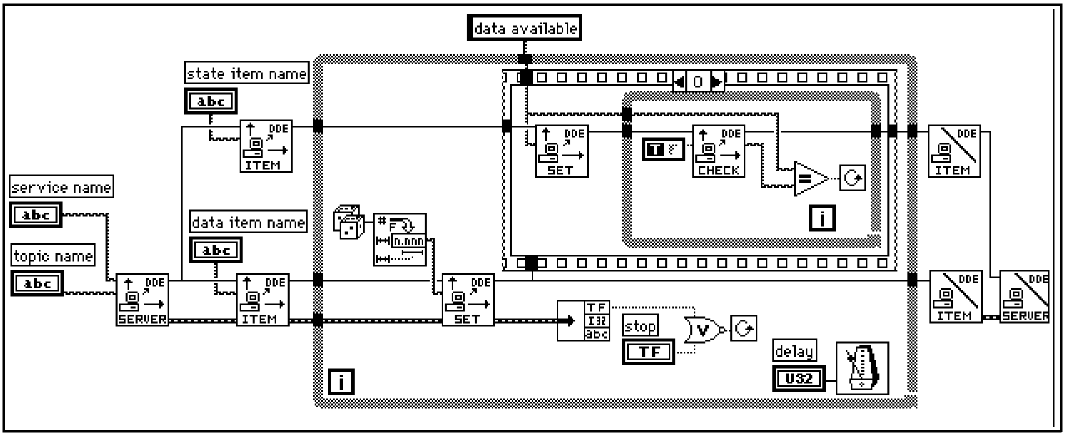
The client server examples in the preceding section work well for monitoring data. However, in these examples there is no assurance that the client receives all the data that the server sends. Even with the DDE Advise loop, if the client does not check for a data change frequently enough, the client can miss a data value that the server provided.

In some applications, missed data is not a problem. For example, if you are monitoring a data acquisition system, missed data may not cause problems when you are observing general trends. In other applications, you may want to ensure that no data is missed.

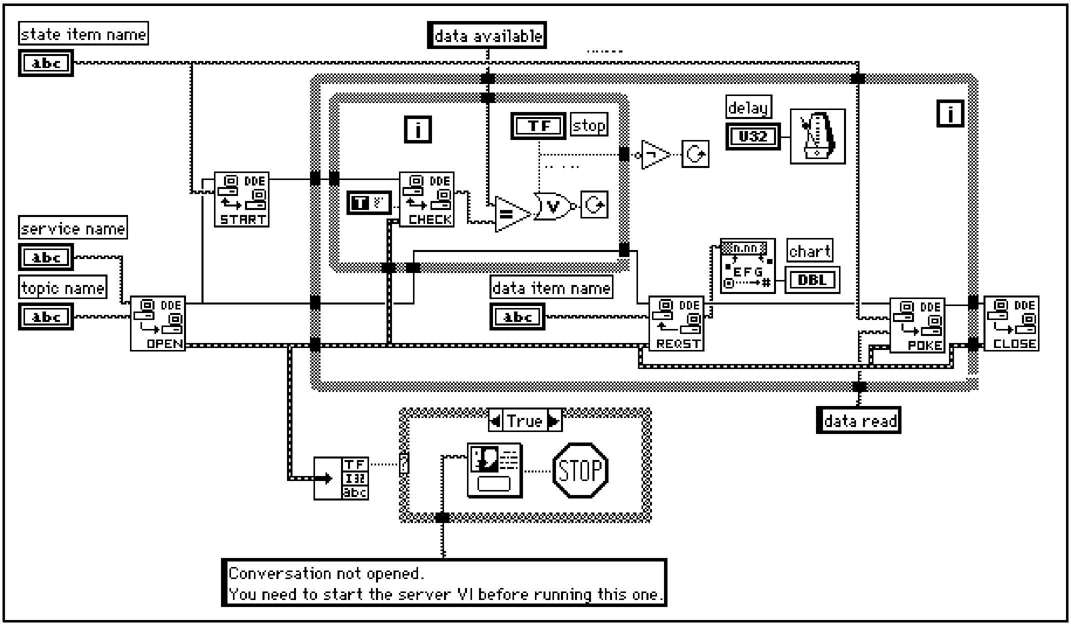
One major difference between TCP and DDE is that TCP queues data so that you do not miss it and you get it in the correct order. DDE does not provide this service.

In DDE, you can set up a separate item, which the client uses to acknowledge that it has received the latest data. You then update the acquired data item to contain a new point only when the client acknowledges receipt of the previous data.

For example, you can modify the server example shown in the *Requesting Data versus Advising Data* section of this chapter to set a *state* item to a specific value after it has updated the acquired data item. The server then monitors the *state* item until the client acknowledges receipt of data. This modification is shown in the following block diagram.



A client for this server, as shown in the following diagram, monitors the state item until it changes to *data available*. At that point, the client reads the data from the acquired data item provided by the server, and then updates the state item to *data read* value.



Conversation not opened.
You need to start the server VI before running this one.

This technique makes it possible to synchronize data transfer between a server and a single client. However, it has some shortcomings. First, you can have only one client. Multiple clients can conflict with one another. For example, one client might receive the data and acknowledge it before the other client notices that new data is available.You can build more complicated DDE diagrams to deal with this problem, but they quickly become awkward. For applications that involve only a single client, this is not a problem.

Another problem with this technique of synchronizing communication is that the speed of your acquisition becomes controlled by the rate at which you transfer data. You can address this issue by breaking the acquisition and the transmission into separate loops. The acquisition can queue data which the transmission loop would send. This is similar to the TCP Server example in which the server handles multiple connections.

If your application needs reliable synchronization of data transfer, you may want to use TCP/IP instead, because it provides queueing, acknowledgment of data transfer, and support for multiple connections at the driver level.

Networked DDE

You can use DDE to communicate with applications on the same computer or to communicate over the network with applications on different computers. To use networked DDE, you must be running Windows for Workgroups 3.1 or greater, Windows 95, or Windows NT. The standard version of Windows 3.1 does not support networked DDE.

Each computer under Windows for Workgroups has a network computer name. You configure this name using the Network control panel.

When you communicate over the network, the meaning of the service and topic strings change. The service name changes to indicate that you want to use networked DDE, and includes the name of the computer you want to communicate with. The service name is of the following form:

```
\\computer-name\ndde$
```

You can supply any arbitrary name for the topic. You then edit the SYSTEM.INI file to associate this topic name with the actual service and topic that will be used on the remote computer. This configuration also includes parameters that configure the network connection. Following is an example of what this section would look like:

```
[DDE Shares]

topicname = appname, realtopic, ,31,,0,,0,0,0
```

The topicname is the name that your client VI uses for the topic. Appname is the name of the remote application. With networked DDE, this must be the same as the service name. Realtopic is the topic to use on the remote computer. The remaining parameters configure the way DDE works. Use the parameters as listed in the preceding example. The meaning of these parameters is not documented by Microsoft.

For example, if you want two computers running LabVIEW to communicate using networked DDE, the server needs to use LabVIEW for the service name, and a name, such as labdata, for the topic.

Assuming the server computer name is Lab, the client tries to open a conversation using the \\Lab\ndde$ for the service. For the topic, the client can use a name of remotelab.

For this to work, you must edit the `SYSTEM.INI` file of the server computer to have the following line in the `[DDEShares]` section:

```
    remotelab=LabVIEW,labdata,,31,,0,,0,0,0
```

For Windows NT, launch `DDEShare.exe`, which is located in the **winnt/system 32** directory. Choose **Shares»DDE Shares…** and then select **Add a Share…** to register the service name and topic name on the server.For more information, see the *Using NetDDE* section of Chapter 52, *DDE VIs*.

OLE Automation (Windows Only)

OLE (Object Linking and Embedding) Automation is a protocol for accessing the functions and methods of one Windows application and making them available for use by another Windows application. OLE Automation works with Windows 95 and Windows NT only, not Windows 3.x.

If an application exposes objects and provides a method of operating on those objects, it is called an *OLE automation server*. Applications that use the methods exposed by another application are *OLE automation clients/controllers*.

LabVIEW contains VIs that enable it to become an automation client. Helper VIs are provided.

AppleEvents (Macintosh Only)

AppleEvents is a Macintosh specific protocol that allows applications to communicate with each other. As with DDE, it is a protocol in which applications use a message to request actions or return information from other applications. An application can send a message to itself, an application on the same computer, or an application running on a computer elsewhere on the network.

You can use AppleEvents to send other commands to other applications, such as open or print, or to send data requests, such as spreadsheet information.

LabVIEW contains VIs for sending some of commands common to most applications. The VIs are easy to use, and do not require detailed knowledge of how AppleEvents work.

These VIs use the low level AESend VI to send AppleEvents. Apple has defined a large *vocabulary* for messages to help standardize AppleEvent communication. You can combine *words* in this vocabulary to build complex messages. You can use this VI to send arbitrary AppleEvents to other applications. However, creating and sending AppleEvents at this level is complicated and requires detailed understanding of AppleEvents. See *Inside Macintosh* and the *AppleEvent Registry*.

Client Server Model

You cannot use the AppleEvent VIs to create LabVIEW diagrams that behave as servers. The VIs are used to send messages to other applications. If you need diagram-based server capabilities, you must use TCP or PPC.

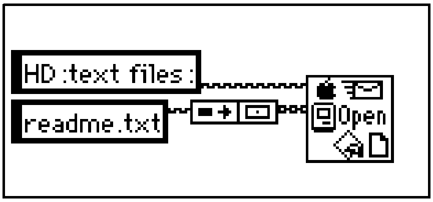
LabVIEW itself acts as an AppleEvent server, in that it understands and responds to a set of AppleEvents. Specifically, using AppleEvents, you can instruct LabVIEW to open VIs, print them, run them, and close them. You can ask LabVIEW whether a given VI is running. You can also tell LabVIEW to quit.

Using these server capabilities, you can instruct other LabVIEW applications to run VIs, and control LabVIEW remotely. You can also command LabVIEW to send messages to itself, instructing the loading of specific VIs. For example, in large applications where memory is limited, you can replace subVI calls with calls to the AESend Open, Run, Close VI to load and run VIs as necessary. Notice that when you run a VI this way its front panel opens, just as if you had selected **File»Open...**.

AppleEvent Client Examples

Launching Other Applications

To send a message to an application, that application must be running. You can use the AESend Finder Open VI to launch another application. This VI sends a message to the Finder. The Finder is, in itself, an application that understands a limited number of AppleEvents. The following simple example shows how you can use AppleEvents to launch Teach Text with a specific text file.



If the application is on a remote computer, then you must specify the location of that computer. You can use inputs to the AESend Finder Open VI to specify the network zone and the server name of the computer with which you want to communicate. If the network zone and server name are not specified, as in the preceding application, they default to those of the current computer.

Notice that if you try to send messages to another computer, you are automatically prompted to log onto that computer. There is no method for avoiding this prompt, because it is built-in to the operating system. This can cause problems when you want your application to run on an unattended computer system.

Sending Events to Other Applications

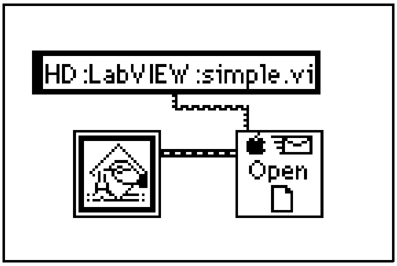
Once an application is running, you can send messages to that application using other AppleEvents. Not all applications support AppleEvents, and those that do may not support every published AppleEvent. To find out which AppleEvents an application supports, consult the documentation that comes with that application.

If the application understands AppleEvents, you call an AppleEvent VI with the Target ID for the application. A Target ID is a cluster that describes a target location on the network (zone, server, and supporting application). You do not need to worry about the exact structure of this cluster because LabVIEW provides VIs that you can use to generate a Target ID.

There are two ways to create a Target ID. You can use the Get Target ID VI to programmatically create a Target ID based upon the application name and network location. Or, you can use the PPC Browser VI, which displays a dialog box listing applications on the network that are aware of AppleEvents. You interactively select from this list to create a Target ID.

You can also use the PPC Browser VI to find out if another application uses AppleEvents. If you run the VI and select the computer that is running the application, the dialog box will list the application if it is AppleEvent aware.

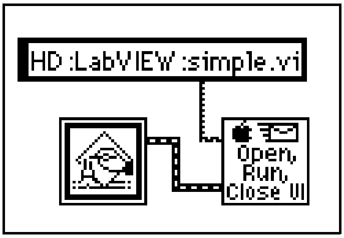
In the following diagram, LabVIEW interactively selects an AppleEvent aware application on the network and tells it to open a document. In this case, LabVIEW is telling the application to open a VI.



Dynamically Loading and Running a VI

The AESend Open, Run, Close VI sends messages asking LabVIEW to run a VI. First, it sends the Open Document Message and LabVIEW opens a VI. Then, the Open Run Close VI sends the LabVIEW Run VI message and LabVIEW runs the specified VI. Next, Open Run Close sends the VI Active? message, and LabVIEW returns the status of a specified VI, until the VI is no longer running. Finally, the VI sends the Close VI message.

Assuming the target LabVIEW is on another computer, you could use the following diagram to load and run the VI. If you are sending it to the current LabVIEW, you do not need the PPC Browser VI.



PPC (Macintosh Only)

Program to Program Communication (PPC) is a Macintosh protocol for transferring blocks of data between applications. You can use it to create VIs that act as clients or servers. Although supported by all Macintoshes running System 7.*x*, it is not commonly used by most Macintosh applications. Instead, most Macintosh applications use AppleEvents, a high-level protocol for sending commands between applications, to communicate.

Although PPC is not as commonly supported as AppleEvents, it does provide some advantages. Because it is at a lower level, it provides better performance than AppleEvents. Also, in LabVIEW you can create VIs that use PPC to act as clients or servers. You cannot create diagrams that act as AppleEvent servers.

PPC is similar in structure to TCP, in terms of both server and client applications. The PPC method for specifying a remote application is different from the TCP method. Other than that, the two protocols provide similar performance and features. Both protocols handle queueing and reliable transmission of data. You can use both protocols with multiple open connections.

In deciding between TCP and PPC, the main point to consider is which platforms you plan to run your VIs on, and with which platforms you will communicate. If your application is Macintosh only, PPC is a good choice, because it is built-in to the operating system. TCP is built-in to Macintosh operating system version 7.5. To use TCP with an earlier system you must buy a separate TCP/IP driver from Apple. If buying the separate driver is not an issue, then you may want to use TCP, because the TCP interface is simpler than PPC. PPC uses some fairly complicated data structures to describe addresses.

If your application must communicate with other platforms or run on other platforms, then you should use TCP/IP.

Ports, Target IDs, and Sessions

To communicate using PPC, both clients and servers must open ports that they use for subsequent communication. The Open Port VI opens the port using a cluster that contains, among other things, the name that you want to use for the port.

Ports are used to distinguish between different services that an application provides. Each application can have multiple ports open simultaneously.

Each port can support several simultaneous sessions or conversations. To open a session, a client uses a Target ID indicating the location of the server. PPC uses the same type of Target ID that the AppleEvent VIs use. You can use the PPC Browser or the Get Target ID VIs to generate the Target ID for the remote application.

A server waits for clients to attempt to open a session by using the PPC Inform Session VI. The server can accept or reject the session by using the PPC Accept Session VI.

A client can attempt to open a session with a server by using the PPC Start Session VI.

After the session is started, you can use the PPC Read and PPC Write VIs to transfer data. You can close a session using PPC End Session, and you can close a port using the PPC Close Port VI.

PPC Client Example

The following discussion explains how you can use PPC to fulfill each component of the general Client model.

Open Conn to Srvr

Use the PPC Open Connection and PPC Open Session VIs to open a connection to a server. This requires that you specify the Target ID of the server, which you can get by using either the PPC Browser VI or the Get Target ID VI. The end result is a port refnum and a session refnum, which are used to communicate with the server.

Exec Cmd on Srvr

To execute a command on the server, use the PPC Write VI to send the command to the server. Next, use the PPC Read VI to read the results from the server. With the PPC Read VI, you must specify the number of characters you want to read. As with TCP, this can be awkward, because the length of the response can vary. The server can have a similar problem, because the length of a command may vary.

Following are several methods for addressing the problem of varying sized commands. These methods can also be used with TCP.

- Precede the command and the result with a fixed size parameter that specifies the size of the command or result. In this case, read the size parameter, and then read the number of characters specified by the size. This option is efficient and flexible.

- Make each command and result a fixed size. When a command is smaller than the size, you can pad it out to the fixed size.
- Follow each command and result with a specific terminating character. To read the data, you then need to read data in small chunks until you get the terminating character.

Close
Conn
to Srvr

Use the PPC Close Session and PPC Close Connection VIs to close the connection to the server.

PPC Server Example

The following discussion explains how you can use PPC to fulfill each component of the general Server.

Initialize
Server

Use PPC Open Port in the initialization phase to open a communication port.

Wait
for a
Conn

Use the PPC Inform Session VI to wait for a connection. With PPC, you can either automatically accept incoming connections, or you can choose to accept or reject the session by using the PPC Accept Session VI. This process of waiting for a session and then approving the session allows you to screen connections.

Wait
for a
Cmd

When a connection is established, you can read from that session to retrieve a command. As was discussed in the *PPC Client Example* section, you must decide the format for commands. If commands are preceded by a length field, then you need to first read the length field, and then read that amount of data.

Exec
Cmd

Execution of a command should be protocol independent, because it is something done on the local computer. When finished, you pass the results to the next stage, where they are transmitted to the client.

Return
Results

Use the PPC Write VI to return the result. As discussed in the *PPC Client Example* section, the data must be formatted in a form that the client can accept.

Close
Conn

Use the PPC Close Session VI to close the connection.

Shut
Down
Server

Finally, when the server is finished, Use the PPC Close Port VI to close the port that you opened in the initialization phase.

PPC Server with Multiple Connections

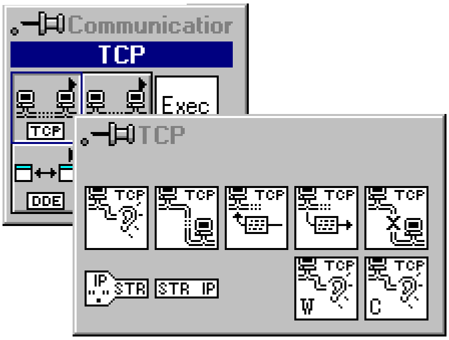
PPC handles multiple sessions and multiple ports easily. The methods for implementing each component of a server, as described in the preceding section, also work for a server with multiple connections.

Chapter 50

TCP VIs

This chapter discusses Internet Protocol (IP), Transmission Control Protocol (TCP), and internet addresses, and describes the LabVIEW TCP VIs. Refer to Chapter 49, *Introduction to LabVIEW Communication VIs and Functions*, for an overview of TCP/IP and examples of TCP client/server applications.

The following illustration shows the **TCP** palette, which you access by selecting **Functions»Communication»TCP**.



For examples of how to use the TCP VIs, see the examples in `examples\comm\tcpex.llb`.

Internet Protocol (IP)

Internet Protocol (IP) performs the low-level service of packaging data into components called *datagrams*. A datagram contains, among other things, the data and a header indicating the source and destination addresses. IP determines the correct path for the datagram to take across the network or internet, and sends the data to the specified destination.

The original host may not know the complete path that the data will take. Using the header, any host on the network can route the data to the destination, either directly or by forwarding it to another host. Because some systems have different transfer capabilities, IP can fragment datagrams into smaller segments as necessary; when the data arrives at the destination, IP automatically reassembles the data into its original form.

IP makes a best-effort attempt to deliver data, but cannot guarantee delivery. Also, because IP routes each datagram separately, they may arrive out of sequence. In fact, IP may deliver a single packet more than once if it is duplicated in transmission. IP does not determine the order of packets. Instead, higher-level protocols layered above IP order the packets and ensure reliable delivery. For this reason, IP is rarely used directly; instead, TCP and UDP, which are built on top of IP, are most often used to transfer information.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) ensures reliable transmission across networks, delivering data in sequence without errors, loss, or duplication. When you pass data to TCP, it attaches additional information and gives the data to IP, which puts the data into datagrams and transmits it. This process reverses at the receiving end, with TCP checking the data for errors, ordering the data correctly, and acknowledging successful transmissions. If the sending TCP does not receive an acknowledgment, it retransmits the data segment.

Using TCP

TCP is a connection-based protocol, which means that sites must establish a connection before transferring data. TCP permits multiple simultaneous connections.

You initiate a connection either by waiting for an incoming connection or by actively seeking a connection with a specified address. In establishing TCP connections, you have to specify both the address and a port at that address. A port is represented by a number between 0 and 65535. With UNIX, port numbers less than 1024 are reserved for privileged applications. Different ports at a given address identify different services at that address, and make it easier to manage multiple simultaneous connections.

You can actively establish a connection with a specific address and port using the TCP Open Connection VI. Using this VI, you specify the address and port with which you want to communicate. If the connection is successful, the VI returns a connection ID that uniquely identifies that connection. Use this connection ID to refer to the connection in subsequent VI calls.

You can use two methods to wait for an incoming connection:

- With the first method, you use the TCP Listen VI to create a listener and wait for an accepted TCP connection at a specified port. If the connection is successful, the VI returns a connection ID and the address and port of the remote TCP.
- With the second method, you use the TCP Create Listener VI to create a listener, and then use the Wait on Listener VI to listen for and accept new connections. Wait on Listener returns the same listener ID that was passed to the VI, as well as the connection ID for a connection. When you are finished waiting for new connections, you can use TCP Close to close a listener. You can not read from or write to a listener.

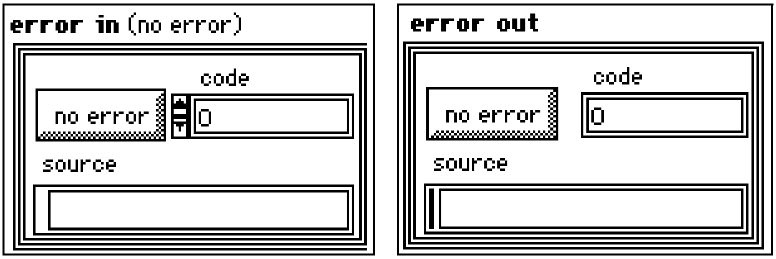
The advantage of using the second method is that you can cancel a listen operation by calling TCP Close. This is useful in the case where you want to listen for a connection without using a timeout, but you want to cancel the listen when some other condition becomes true (for example, when the user presses a button).

When a connection is established, you can read and write data to the remote application using the TCP Read and TCP Write VIs.

Finally, use the TCP Close Connection VI to close the connection to the remote application. Note that if there is unread data and the connection closes, that data may be lost. Connected parties should use a higher level protocol to determine when to close the connection. Once a connection is closed, you may not read or write from it again.

TCP Errors

The TCP VIs report errors in clusters as the following illustration shows. See the *Error Codes* manual for a list of the TCP error codes and their descriptions.

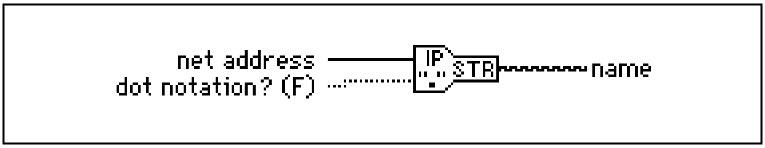


TCP VI Descriptions

The following TCP VIs are available.

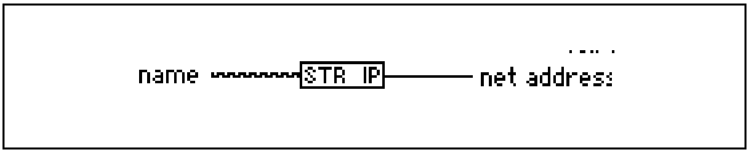
IP To String

Converts an IP network address to a string.



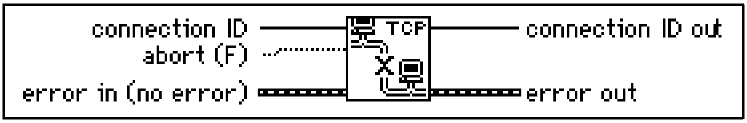
String To IP

Converts a string to an IP network address.



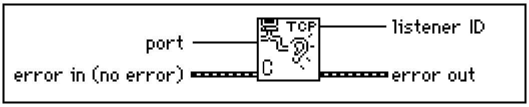
TCP Close Connection

Closes the connection associated with **connection ID**.



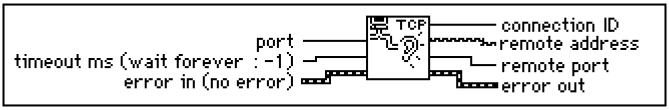
TCP Create Listener

Creates a listener for a TCP connection.



TCP Listen

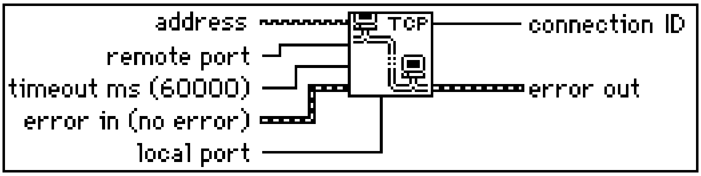
Creates a listener and waits for an accepted TCP connection at the specified port.



When a listen on a given port begins, you may not use another TCP Listen VI to listen on the same port. For example, suppose a VI has two TCP Listen VIs on its block diagram. If you start a listen on port 2222 with the first TCP Listen VI, any attempts to listen on port 2222 with the second TCP Listen VI fail.

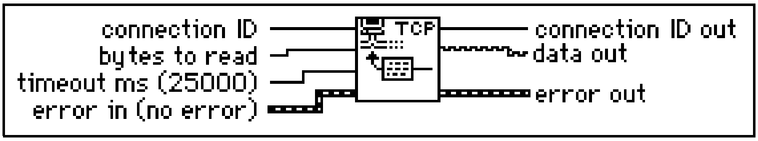
TCP Open Connection

Attempts to open a TCP connection with the specified address and port.



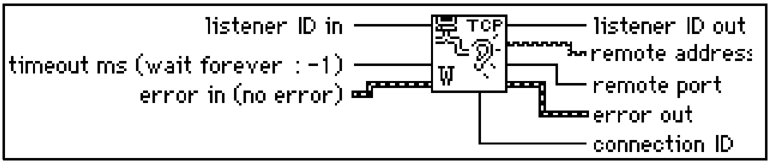
TCP Read

Receives up to **bytes to read** bytes from the specified TCP connection, returning the results in **data out**.



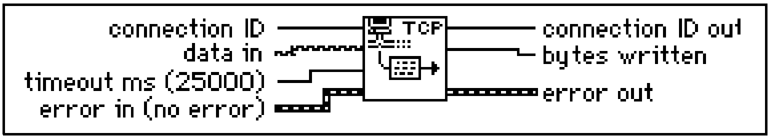
TCP Wait on Listener

Waits for an accepted TCP connection at the specified port.



TCP Write

Writes the string **data in** to the specified TCP connection.

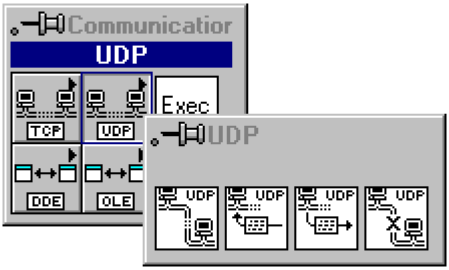


Chapter 51

UDP VIs

This chapter describes a set of VIs that you can use with User Datagram Protocol (UDP), a protocol in the TCP/IP suite for communicating across a single network or an interconnected set of networks.

The following illustration shows the **UDP VI** palette, which you access by selecting **Functions»Communication»UDP**.



UDP Overview

UDP transmits data across networks. UDP can communicate to specific processes on a computer. When a process opens a network connection to a particular port it only receives datagrams that are addressed to that port on that computer. When a process sends a datagram, it must specify the computer and port as the destination.

There are several reasons why UDP is rarely used directly. UDP does not guarantee data delivery. Each datagram is routed separately, so datagrams may arrive out of order, be delivered more than once or not delivered at all.

Typically, UDP is used in applications where reliability is not critical. For example, an application might transmit informative data to a destination frequently enough that a few lost segments of data are not problematic.

Using UDP

UDP is not a connection-based protocol like TCP. This means that a connection does not need to be established with a destination before sending or receiving data. Instead, the destination for the data is specified when each datagram is sent. The system does not report transmission errors.

You can use the UDP Open VI to create a connection. A port must be associated with a connection when it is created so that incoming data can be sent to the appropriate application. The number of simultaneously open UDP connections depends on the system. UDP Open returns a Network Connection refnum, an opaque token used in all subsequent operations pertaining to that connection.

You can use the UDP Write VI to send data to a destination and the UDP Read VI to read it. Each write requires a destination address and port. Each read contains the source address and port. Packet boundaries are preserved. That is, a read never contains data sent in two separate write operations.

In theory, you should be able to send data packets of any size. If necessary, a packet is disassembled into smaller pieces and sent on its way. At their destination, the pieces are reassembled and the packet is presented to the requesting process. In practice, systems only allocate a certain amount of memory to reassemble packets. A packet that cannot be reassembled is thrown away. The largest size packet that can be sent without dissassembly depends on the network hardware.

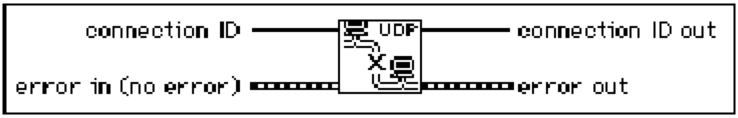
When LabVIEW finishes all communications, calling the UDP Close VI frees system resources.

UDP VI Descriptions

The following UDP VIs are available.

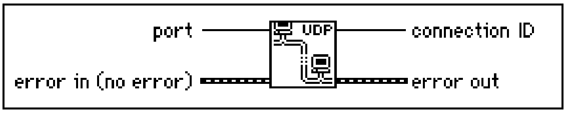
UDP Close

Closes the UDP connection specified by **connection ID**.



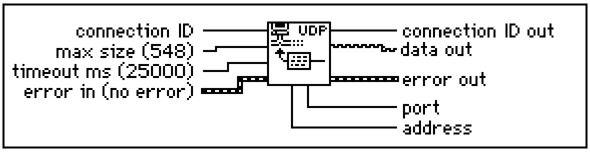
UDP Open

Attempts to open a UDP connection on the given **port**. **Connection ID** is an opaque token used in all subsequent operations relating to the connection.



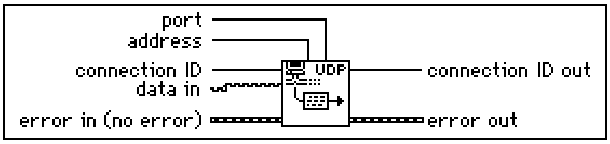
UDP Read

Returns a datagram in the string **data out** that has been received on the UDP connection specified by **connection ID**.



UDP Write

Writes the string **data in** to the remote UDP connection specified by **address** and **port**.

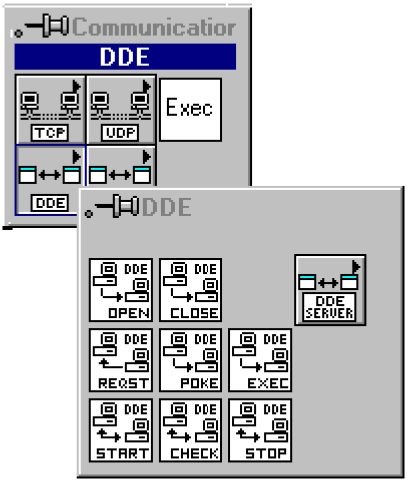


Chapter 52

DDE VIs

This chapter describes the LabVIEW VIs for Dynamic Data Exchange (DDE) for Windows 3.1, Windows 95, and Windows NT. These VIs execute DDE functions for sharing data with other applications that accept DDE connections.

The following illustration shows the DDE VI palette, which you access by selecting **Functions»Communication»DDE**.



The **DDE** palette includes the **DDE Server** subpalette.

For examples of how to use the DDE VIs, see the examples in `examples\comm\DDEexamp.llb`.

DDE Overview

DDE is a *client-controlled* message passing protocol. One application, the *client*, passes messages to another application, the *server*.

Both applications must be running, and both must give Windows their callback function address before DDE communication can begin. The callback function accepts any DDE messages that Windows sends to the application.

The client initiates a conversation with the server by sending a DDE connect message. After establishing the conversation, the client can send commands or data to the server, or request data from the server.

A client can request data from a server by a *request* or an *advise*. A request is a single transfer of data. An advise establishes an active link between the two applications. The server then informs the client every time the advise value changes. When the client no longer needs the changed values, it sends an advise stop message to the server.

When all the DDE communication for the conversation is complete, the client sends a close conversation message to the server.

Using DDE as a Client

The Dynamic Data Exchange VIs give LabVIEW full DDE client capability.

To use DDE, you must first establish a conversation using the DDE Open Conversation VI. The VI must specify the *service* and the *topic*. The service usually corresponds to the name of the server application and the topic to the active file. DDE messages then carry data to or from specific locations in the active file. For more information on how a specific application handles topic names and data item locations, consult the documentation for that application.

When you have established a conversation, you can send data using the DDE Poke VI, send commands using the DDE Execute VI, obtain data with the DDE Request VI, or initiate an advise protocol with the DDE Advise Start VI.

The DDE Request VI sends a DDE message to the server every time you call it. The server must then check the data requested and return it in another DDE message. If your VI checks the value frequently, an advise protocol might be more efficient than a request.

The DDE Advise Start VI creates a local copy of the data value you are interested in. When you call the DDE Advise Check VI, the VI returns this value without sending any DDE messages. At the same time, the server application sends DDE messages every time the value changes, so that the local value is always current. If the value seldom changes but is often needed, an advise can significantly reduce the required number of DDE messages.

Caution: ***During a conversation, you must pass the conversation refnum to all other DDE VIs involved in that conversation. Windows uses these refnums to identify the conversation. If you alter the conversation refnum, or do not specify or wire the conversation refnum, the VI will fail. The same is true for the advise refnum. If you alter advise refnum, or do not specify or wire advise refnum for the DDE Advise Check VI or the DDE Advise Stop VI, the VIs will fail and may cause a system failure.***

The DDE protocol used by LabVIEW is ASCII based, and the transmission is terminated when a null byte is reached. If the binary data has a null byte (00) in it, the transmission will end.

To send a number to another application, you must convert that number to a string. In the same way, you must convert numbers received through a request or advise from the string format. Use the conversion VIs from **Functions»String**. See Chapter 6, *String Functions*, earlier in this manual for further information on how to use string conversion VIs.

Stop all advises and close all conversations using DDE Advise Stop and DDE Close Conversation after all DDE commands have executed. This releases the system resources associated with these VIs.

Using DDE as a Server

The first step to becoming a DDE server is to use the DDE Srv Register Service VI to tell Windows what your service name and topic are going to be. At this point other applications can open DDE conversations with your service.

You can call the DDE Srv Register Service VI multiple times with different service names to establish multiple services or multiple times with the same service name but different topic names to establish multiple topics for one service.

After specifying your service and topic names, you can define items for that service using the DDE Srv Register Item VI. After this call, other applications can request or poke the item, as well as initiate advises on that item. LabVIEW fully manages all these transactions.

To change the value of an item, call the DDE Srv Set Item VI. This VI changes the value and informs all clients that have advises on them.

To monitor whether a client has changed an item with a poke, call the DDE Srv Check Item VI. This VI either returns the current value immediately or waits until a client changes the value. If a client pokes the value before DDE Srv Check Item is called with **wait for poke** true, DDE Srv Check Item returns immediately and reports that the value was poked.

You call the DDE Srv Unregister Item VI and the DDE Srv Unregister Service VI to close down your DDE server when you are finished. LabVIEW automatically disconnects any client conversations connected to your server when DDE Srv Unregister Service is called.

Using NetDDE

NetDDE is built into Windows for WorkGroups 3.11, Windows 95 and Windows NT. It is also available for Windows 3.1 with an add-on package from WonderWare. If you are using Windows 3.1 with the WonderWare package, consult the WonderWare documentation on how to use NetDDE.

If you are using Windows for WorkGroups, Windows 95, or Windows NT, use the following instructions:

SERVER MACHINE

Windows for Workgroups

Add the following line to the `[DDE Shares]` section of the file `system.ini` on the server (application receiving DDE commands):

```
lvdemo = service_name,topic_name,,31,,0,,0,0,0
```

where

`lvdemo` can be any name.

`service_name` is typically the name of the application, such as `excel`.

`topic_name` is typically the specific file name, such as `sheet1`.

Enter other commas and numbers as shown.

Windows 95

☞ **Note:** ***NetDDE is not automatically started by Windows 95. You need to run the program\WINDOWS\NETDDE.EXE. (This can be added to the startup folder so that it is always started.)***

To set up a NetDDE server on Windows 95:

- Run`\WINDOWS\REGEDIT.EXE`.
- In the tree display, open the folder `My Computer\HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NetDDE\DDE Shares`.
- Create a new DDE Share by selecting **Edit»New»Key** and give it the name `lvdemo`.
- With the `lvdemo` key selected, add the required values to the share as follows. (For future reference, these keys are just being copied from the `CHAT$` share but `REDEGIT` does not allow you cut, copy, or paste keys or values.) Use **Edit»New** to add new values. When you create the key, there will a default value named (`Default`) and a value of (`value not set`). Leave these values alone and add the following:

Table 52-1. Values to Add in Place of Default

Value Type	Name	Value
Binary	Additional item count	`00 00 00 00`
String	Application	`service_name`
String	Item	`service_name`
String	Password1	`service_name`
String	Password2	`service_name`
Binary	Permissions1	`1f 00 00 00`
Binary	Permissions2	`00 00 00 00`
String	Topic	`topic_name`

- Close `REGEDIT`.
- Restart the machine. (NetDDE must be restarted for changes to take affect.)

Windows NT

Launch `DDEShare.exe`, found in the `winnt\system32` directory. Select from the **Shares»DDE Shares»Add a Share...** to register the service name and topic name on the server.

CLIENT MACHINE

On the client machine (application initiating DDE conversation) no configuration changes are necessary.

Use the following inputs to `DDE Open Conversation.vi`:

Service: `\\machine_name\ndde$`

Topic: `lvdemo`

where:
`machine_name` specifies the name of the server machine

`lvdemo` matches the name specified in the `[DDE Shares]` section on the server.

Consider the examples `Chart Client.vi` and `Chart Server.vi` found in `examples\network\ddeexamp.llb`. To use those VIs to pass information between two computers using NetDDE, you should do the following:

Server Machine:

1. Do not modify any front panel values.
2. In the `system.ini` file of the Server machine, add the following line in the `[DDEShares]` section:

```
lvdemo = TestServer,Chart,,31,,0,,0,0,0
```

Client Machine:

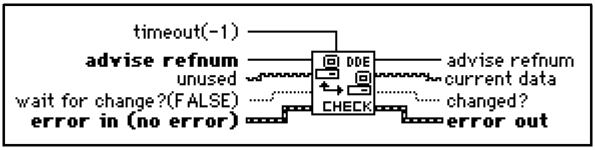
On the front panel, set the controls to the following:
Service = `\\machine_name\ndde$`
Topic = `lvdemo`
Item = `Random`

DDE Client VI Descriptions

The following DDE Client VIs are available.

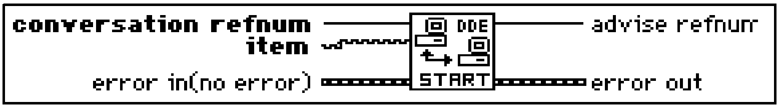
DDE Advise Check

Checks an advise value previously established by DDE Advise Start.



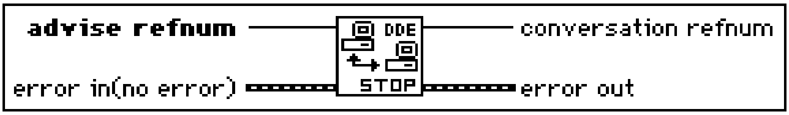
DDE Advise Start

Initiates an advise link.



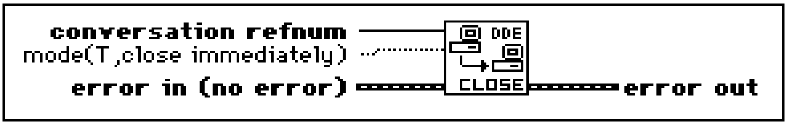
DDE Advise Stop

Cancels an advise link, previously established by DDE Advise Start.



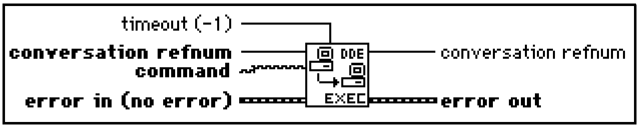
DDE Close Conversation

Closes a DDE conversation.



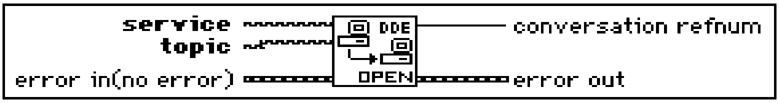
DDE Execute

Tells the DDE server to execute **command**.



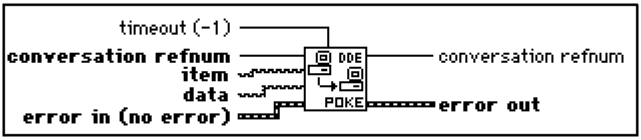
DDE Open Conversation

Establishes a connection between LabVIEW and another application. You must call this VI before you use any other DDE VIs (except Server VIs).



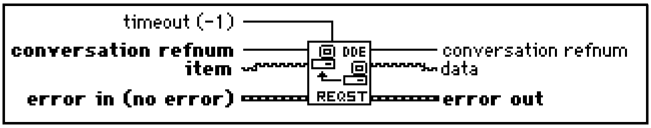
DDE Poke

Tells the DDE server to put the value **data** at **item.**



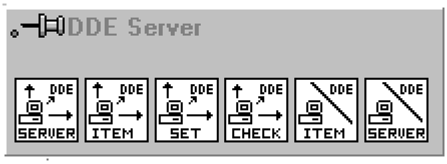
DDE Request

Initiates a DDE message exchange to obtain the current value of **item**.



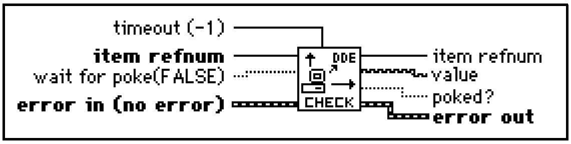
DDE Server VI Descriptions

You access by selecting **Functions»Communication»DDE»DDE Server.**



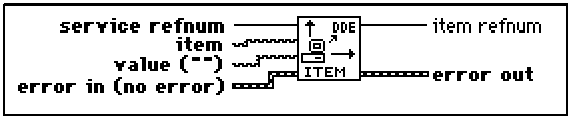
DDE Srv Check Item

Sets the value of a previously defined DDE Item.



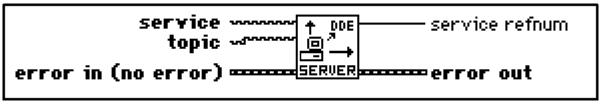
DDE Srv Register Item

Establishes a DDE item for the service specified by **service refnum**.



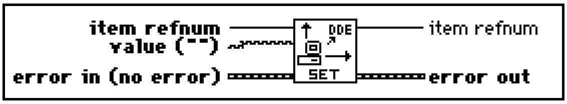
DDE Srv Register Service

Establishes a DDE service to which clients can connect.



DDE Srv Set Item

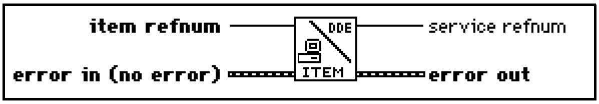
Sets the value of a previously defined DDE Item.



DDE Srv Unregister Item

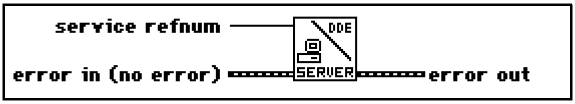
Removes the specified item from its service.

☞ **Note:** ***DDE clients can no longer access the item after this VI completes.***



DDE Srv Unregister Service

Removes the specified service. DDE clients can no longer connect to this service and all current conversations will be closed.



Chapter 53

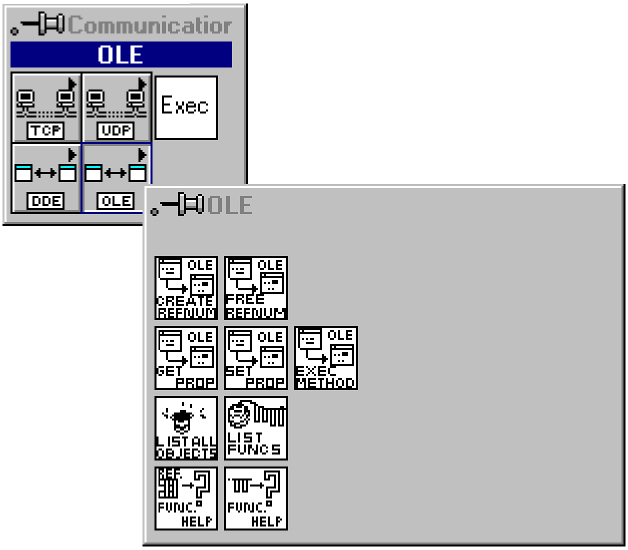
OLE Automation VIs

This chapter discusses the LabVIEW VIs for Object Linking and Embedding (OLE) Automation, a feature which allows LabVIEW to access objects exposed by automation servers in the system.

The OLE Automation VI Library contains two levels of VIs. VIs that are available on the Communication palette represent the higher level of functionality. These VIs use lower level VIs which are hidden from the user, providing for a higher level of encapsulation. Helper VIs are provided.

☞ **Note:** ***These VIs work under Windows NT and Windows 95 only.***

The following illustration shows the **OLE Automation VI** palette, which you access by selecting **Functions»Communication»OLE**.



For examples of how to use the OLE Automation VIs, see the examples in `examples\comm\OLE-xxx.llb`.

OLE Automation Concepts

In the context of Object Linking and Embedding, objects are defined as data abstractions exported by an application. You can manipulate these objects by using another Windows application. Linking and Embedding are two of the methods used to access OLE objects.

You can use OLE Automation to make the functions and methods of one application available for use by other applications. You then access these functions or methods, which are usually grouped into objects.

An application supports automation either as a server or a client. Applications that expose objects and provide methods for operating on those objects are called *OLE automation servers*. Applications that use the methods exposed by another application are called *OLE automation clients/controllers*. The OLE VIs enable LabVIEW to become an automation client.

Using LabVIEW to Implement OLE Automation

An OLE object exposes both methods and properties. Methods have the ability to modify a wide range of values, whereas properties can set or get the value of a specific characteristic of the object. Some servers provide a type library listing all exposed objects and the methods and properties of each object.

The typical steps in creating a client application using C are as follows:

- Get the IDispatch interface of the Object whose methods you want to access.
- Get the DispatchID of the method of that object.
- Invoke the method using the Invoke functions of the IDispatch interface, packing all parameters into the parameter list.

In LabVIEW, do as follows:

- Use the Create Automation Refnum VI to get an Automation refnum which uniquely defines the IDispatch interface.
- Use the Execute Method VI to execute a method belonging to that object. If there is just one parameter, it can be flattened. The type descriptors and the flattened string are then passed in as input parameters. If there are multiple outputs, they are bundled in a

cluster. The resultant cluster is then flattened and wired to the correct input of the VI.

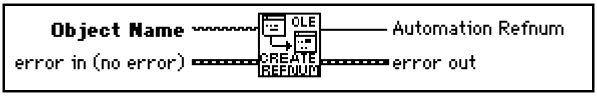
The implementation uses DLLs to perform the actual OLE calls. Parameters are passed to these DLLs as flattened data.

OLE Automation VI Descriptions

The following OLE Automation VIs are available.

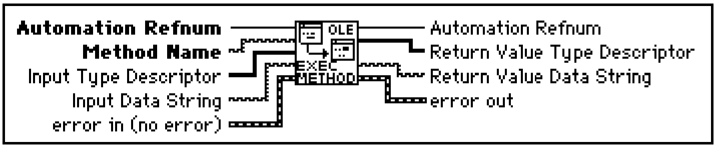
Create Automation Refnum

Given the object name (registered class name) of an OLE object, returns an Automation Refnum uniquely identifying the instantiation.



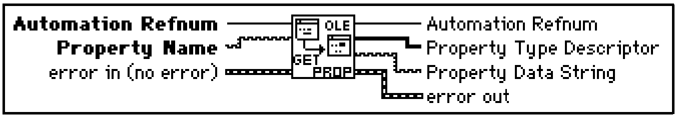
Execute Method

Executes a method.



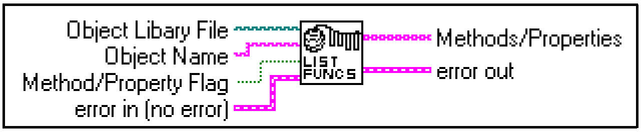
Get Property

Gets the value of a property.



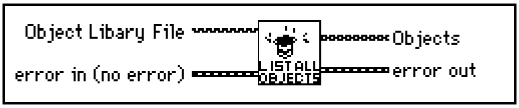
List Methods or Properties

Lists all the methods or properties of an object.



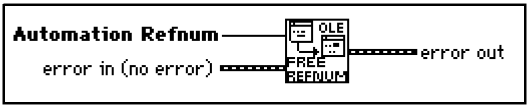
List Objects in Type Library

Lists all the objects in a type library.



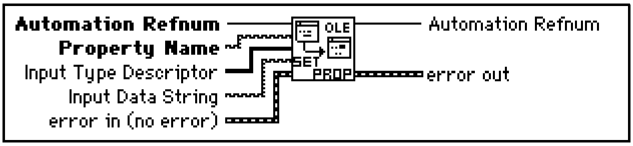
Release Refnum

Releases the refnum passed in as input.



Set Property

Sets the value of a property.



Chapter 54

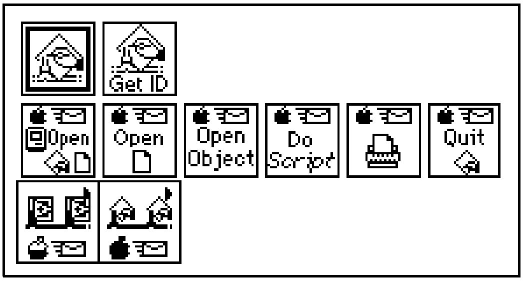
AppleEvent VIs

This chapter discusses the LabVIEW VIs for interapplication communication (IAC), a feature of Apple Macintosh system software version 7 by which Macintosh applications can communicate with each other. You can use LabVIEW with two forms of IAC, AppleEvents and program-to-program communication (PPC).

AppleEvents are a high-level method of communication in which applications use messages to request other applications to perform actions or return information. An application can send these messages to itself, other applications on the same machine, or other applications located anywhere on a network. Apple has defined a large *vocabulary* for messages to help standardize this form of interapplication communication. You can combine *words* in this vocabulary to form very complex messages. This vocabulary is described in detail in the *AppleEvent Registry,* a document available from Apple. Most applications written for System 7, including LabVIEW, respond to some subset of AppleEvents.

PPC is a low-level form of IAC by which applications send and receive blocks of data. PPC provides higher performance than AppleEvents, because the overhead required to transmit information is lower. However, because PPC does not define what kinds of information you can transfer, many applications do not support it. PPC is the best way to send large amounts of information between applications that support PPC. See Chapter 55, *Program to Program Communication VIs*, for more information about PPC.

The following illustration shows the **AppleEvent VI** palette, which you access by selecting **Functions:Communication:AppleEvent**.



☞ **Note:** ***For applications to communicate with IAC, the computer must use system software version 7.0 or greater with Program Linking enabled.***

For examples of how to use the AppleEvent VIs, see the examples located in `examples\comm\AE Examples.llb`.

AppleEvents

LabVIEW can send and respond to AppleEvents. You can use AppleEvent VIs to send AppleEvents. LabVIEW responds to two types of AppleEvents: LabVIEW-defined events and a subset of standard AppleEvents. See the *Sending AppleEvents* section of this chapter for more information.

Some of the ways you can use AppleEvents in LabVIEW applications are listed on the following page:

- You can command LabVIEW to tell another application (even an application on another computer connected by a network) to perform an action. For example, LabVIEW can tell a spreadsheet program to create a graph. See the *Sending AppleEvents* section in this chapter for details.
- You can use a program such as HyperCard as a front end to instruct LabVIEW to run specific VIs.
- You can communicate with and control LabVIEW applications on other machines connected by a network by sending them instructions to perform specific operations. See the *Sending AppleEvents* section in this chapter for details.
- You can command LabVIEW to send messages to itself, instructing itself to load, run, and unload specific VIs. For example, in large

applications where memory is tight, you can replace subVI calls with a utility VI (the AESend Open, Run, Close VI) and dynamically load, run, and unload the VIs. See the *Sending AppleEvents* section in this chapter for details.

The following sections describe in detail how LabVIEW sends and receives AppleEvents.

Sending AppleEvents

The **Communication** subpalette of the **Functions** palette contains VIs for sending AppleEvents. With these VIs, you can select a target application for an AppleEvent, create AppleEvents, and send the AppleEvents to the target application.

The **AppleEvent VIs** palette of the **Communication** subpalette contains VIs that send specific AppleEvent messages. These VIs let you send several standard AppleEvents (Open Document, Print Document, and Close Application) and all the LabVIEW custom AppleEvents. These high-level VIs require little understanding of AppleEvent programming details. Their diagrams also serve as good examples of how to create and send AppleEvents.

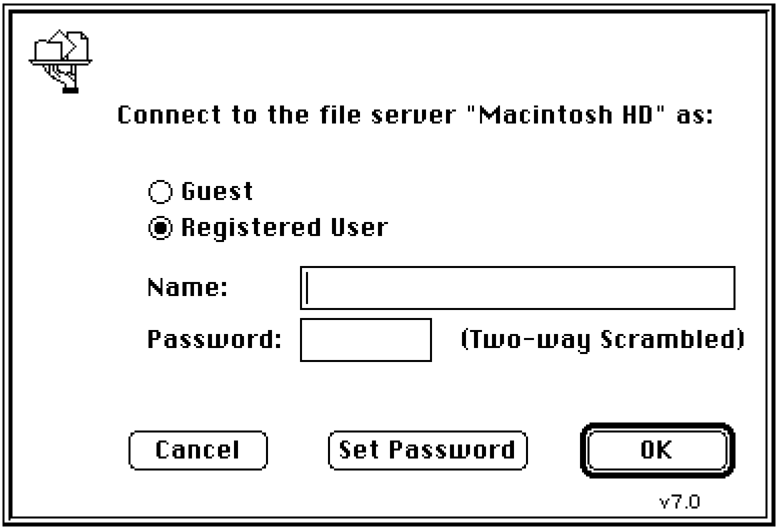
You can use the low-level AESend VI if you want to send an AppleEvent for which LabVIEW provides no VI. The **AppleEvent VIs** palette of the **Communication** subpalette also contains VIs that can help you create an AppleEvent. However, creating and sending an AppleEvent at this level requires detailed understanding of AppleEvents as described in *Inside Macintosh, Volume VI* and the *AppleEvent Registry*.

General AppleEvent VI Behavior

When sending an AppleEvent, you must specify the *target* application for the event. To receive the AppleEvent, the target application must be open. You can use the AESend Finder Open VI to open an application.

The User Identity Dialog Box

Before you send an AppleEvent to another computer, you must use the Users & Groups control panel utility on the destination computer to set up a user name and password for yourself. The first time you send an AppleEvent to an application or Finder on the destination computer, a dialog box prompts you to enter your name and password. The system compares this information to the configuration of the Users & Groups control panel utility on the destination computer.



Connect to the file server "Macintosh HD" as:

○ Guest
● Registered User

Name:
Password: (Two-way Scrambled)

The current design of the AppleEvent Manager does not include a programmatic method for bypassing this dialog box, so you should take this into account when designing VIs that use IAC. For example, you cannot command an unattended remote computer to send an AppleEvent to a third computer; someone must enter user information into the User Identity Dialog Box that appears on the remote computer. The PPC VIs allow for *unauthenticated* sessions if guest access is enabled on the computer with which you wish to communicate, so you may find the PPC VIs more useful for certain kinds of LabVIEW-to-LabVIEW communication.

Target ID

Most VIs that send AppleEvents need a description of the target application that will receive the AppleEvent. The **target ID** is a complex cluster of information, defined by Apple Computer Inc., describing the target application and its location. The following VIs generate the **target ID**, so you do not need to create this cluster on the diagram.

- PPC Browser creates the **target ID** by displaying a dialog box by which you interactively select AppleEvent-aware applications on the network.

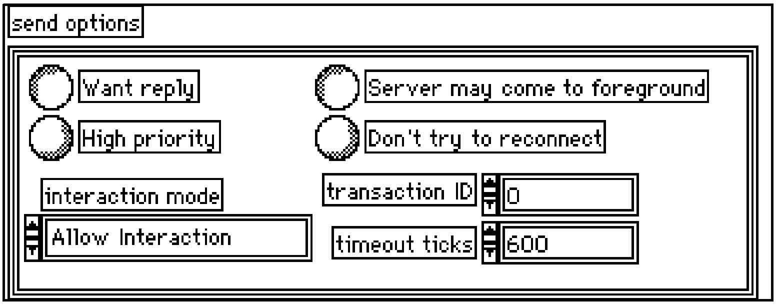
- Get Target ID creates the **target ID** programmatically based on the application's name and network location.

These VIs are discussed in more detail in the *Targeting VIs* section of this chapter.

You need to look at the **target ID** cluster only if you want to pass target information from one VI to another. To create a **target ID** cluster for the front panel of a VI that passes target information to another VI or to an AppleEvent, you can copy the **target ID** cluster from the front panel of one of the AppleEvent VIs.

Send Options

Many of the VIs that send an AppleEvent have a **send options** input, which specifies whether the target application can interact with the user and the length of the AppleEvent timeout.


send options

○ Want reply

○ High priority

○ Server may come to foreground

○ Don't try to reconnect

interaction mode: Allow Interaction

transaction ID: 0

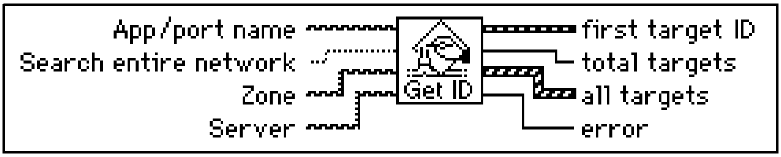
timeout ticks: 600

Targeting VI Descriptions

The following Targeting VIs are available.

Get Target ID

Returns a target ID for a specified application based on its name and location. You can either specify the application's name and location or the VI searches the entire network for the application.

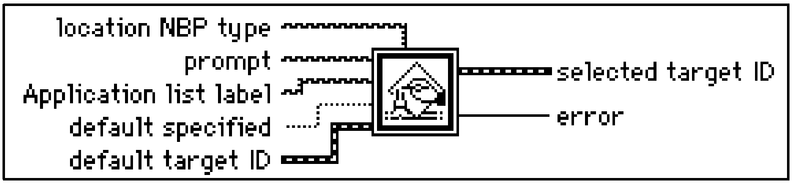


The following table summarizes the operation of **Search entire network**, **Zone**, and **Server**:

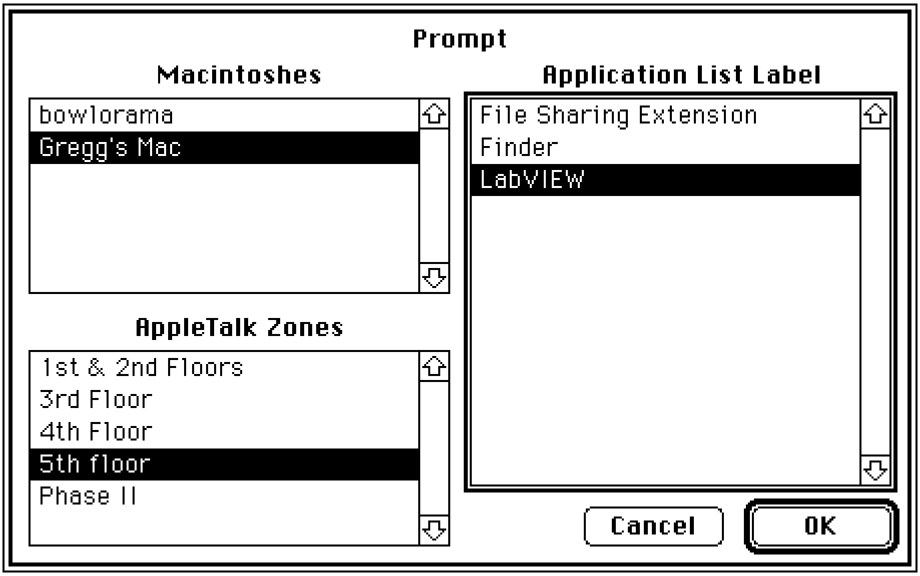
To search the following locations:	Use the following parameters:
The current computer	**Zone** and **Server** must be unwired. Search entire network must be FALSE.
A specific computer on the network	**Zone** and **Server** must specify the target computer's zone and server. (If you do not wire **Zone**, the VI searches the current zone.) **Search entire network** must be FALSE.
A specific zone	**Zone** must specify the zone to be searched. **Server** must be unwired. **Search entire network** must be FALSE.
The entire network	**Search entire network** must be TRUE. The VI ignores **Zone** and **Server**.

PPC Browser

Invokes the PPC Browser dialog box for selecting an application on a network or on the same computer.



You can use this standard Macintosh dialog box to select a zone from the network, an object in that zone (in System 7, this is typically the name of a person's computer), and an application. The VI then returns the **target ID** cluster.



Prompt

Macintoshes

bowlorama
Gregg's Mac

AppleTalk Zones

1st & 2nd Floors
3rd Floor
4th Floor
5th floor
Phase II

Application List Label

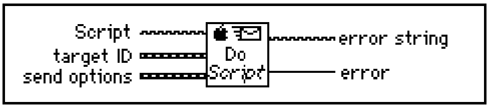
File Sharing Extension
Finder
LabVIEW

AppleEvent VI Descriptions

The following AppleEvent VIs are available.

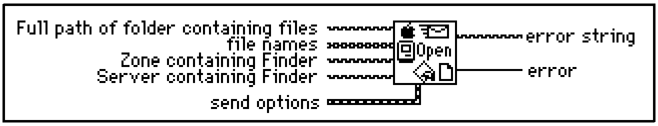
AESend Do Script

Sends the Do Script AppleEvent to a specified target application.



AESend Finder Open

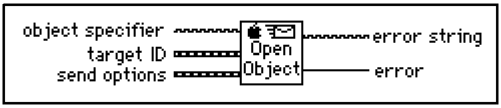
Sends the AppleEvent to open specified applications or documents to the System 7 Finder on the specified machine.



☞ **Note:** ***Apple may change the set of AppleEvents to which the Finder responds so that they more closely conform to the standard set of AppleEvents. As a result, the AppleEvent that AESend Finder Open sends to the Finder may not be supported in future versions of the system software.***

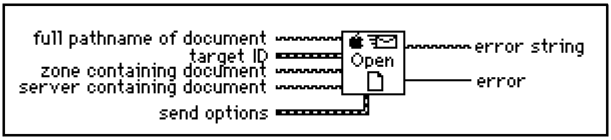
AESend Open

Sends the Open AppleEvent to a specified target application.



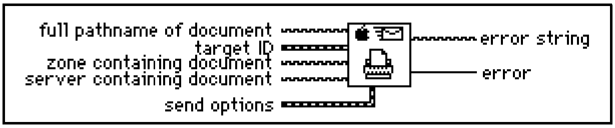
AESend Open Document

Sends the Open Document AppleEvent to the specified target application, telling the application to open the specified document.



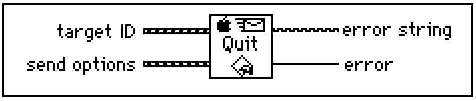
AESend Print Document

Sends the Print Document AppleEvent to the specified target application, telling the application to print the specified document.



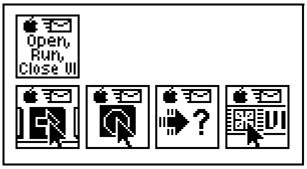
AESend Quit Application

Sends the Quit Application AppleEvent to a specified target application.



LabVIEW Specific AppleEvent VIs

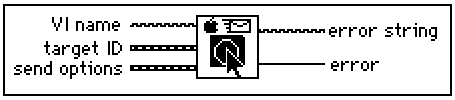
LabVIEW specific AppleEvent VIs send messages that only LabVIEW applications (standard and run-time systems) recognize. To access the LabVIEW Specific Apple Events VIs, select **Functions**:**Communication: LabVIEW Specific Apple Events**.



You should use these VIs only when communicating with LabVIEW applications. You can send these messages either to the current LabVIEW application or to a LabVIEW application on a network. See the *AppleEvent Error Codes* section of the *Error Codes* manual for error information.

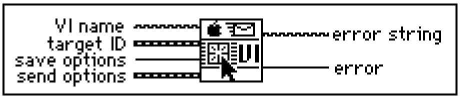
AESend Abort VI

Sends the Abort VI AppleEvent to the specified target LabVIEW application.



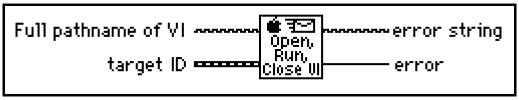
AESend Close VI

Sends the Close VI AppleEvent to the specified target LabVIEW application.



AESend Open, Run, Close VI

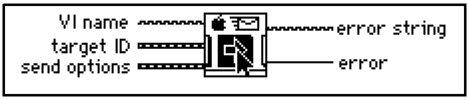
Uses the Open Document, Run VI, VI Active?, and Close VI AppleEvent VIs to make a specified LabVIEW application open, run, and close a VI.



For this VI, you must specify the complete pathname of the VI you want to run. See Chapter 13, *Path Controls and Refnums*, of your *LabVIEW User Manual* for a description of path controls and indicators available in the Controls palette.

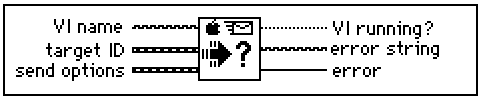
AESend Run VI

Sends the Run VI AppleEvent to the target LabVIEW application.



AESend VI Active?

Sends the VI Active? AppleEvent to the specified target LabVIEW application. **VI running?** is a Boolean indicating whether the VI is currently executing.



Advanced Topics

This section describes some of the advanced programming you can do with AppleEvent VIs.

Constructing and Sending Other AppleEvents

In addition to VIs that send common AppleEvents, you can use lower-level VIs to send any AppleEvent. Using these VIs requires more knowledge of AppleEvents than using the VIs described earlier in this chapter. If you are interested in using these VIs, you should be familiar with the discussion of AppleEvents in *Inside Macintosh, Volume VI,* and the *AppleEvent Registry*.

When sending an AppleEvent, you must include several pieces of information. The event class and event ID identify the AppleEvent you are sending. The event class is a four-letter code which identifies the AppleEvent group. For example, an event class of `core` identifies an AppleEvent as belonging to the set of core AppleEvents. The event ID is another four-letter code that identifies the specific AppleEvent that you wish to send. For example, `odoc` is the four-letter code for the Open Documents AppleEvent, one of the core AppleEvents. To send an AppleEvent using the AESend VI, concatenate the event class and event ID together as an eight-character string. For example, to send the Open Documents AppleEvent, pass the AESend VI the eight-character code `coreodoc`.

If you are sending the AppleEvent to another application, you have to specify **target ID** and **send options**, as described earlier in this chapter.

You can also specify an array of parameters if the target application needs additional information to execute the specified AppleEvent. Because the data structure for AppleEvent parameters is inconvenient for use in LabVIEW diagrams, the AESend VI accepts these parameters as ASCII strings. These strings must conform to the grammar described in the next section. You can use this grammar to describe any AppleEvent parameter. The AESend VI interprets this string to create the appropriate data structure for an AppleEvent, and then sends the event to the specified target.

Creating AppleEvent Parameters

In many cases, an AppleEvent parameter is a single value; however, it can be quite complex, with a hierarchical structure containing components that in turn can contain other components. In LabVIEW, a parameter is constructed as a string, which has a simple grammar with which you can describe all kinds of data that an AppleEvent parameter can be, including complex structures.

An AppleEvent parameter string begins with a keyword, a four-letter code describing the parameter's meaning. For example, if the parameter is a direct parameter (one of the most common types of parameters) you must specify that the keyword is a `keyDirectObject` by using the four-letter code `----` (four dashes). Other examples of keywords include `savo`, short for save options, which is used when sending the Close VI AppleEvent to LabVIEW. Documentation detailing an application's supported AppleEvents should indicate the keywords used for each parameter. See the *Sending AppleEvents to LabVIEW from Other*

Applications section of this chapter for a list of the AppleEvents that you can use with LabVIEW.

Following the keyword, you must specify the parameter data as a string. You can use AppleEvents with many different data types, including strings and numbers. When you specify the data string, the AESend VI converts it to a desired data type based upon the way the data is formatted and optional directives that can be embedded in the string. Each piece of data has a four-letter type code associated with it, indicating its data type. The target application uses this code to interpret the data. For example, if comma-separated items are enclosed in brackets, a list of *AE Descriptors* is created, and the list has a data type of `list`; each of the comma-separated items could in turn be other items, including lists.

You can use a number of VIs in the **AppleEvents VI** palette to create some of the more common parameter strings, including aliases, which are used when referencing files in parameters, and descriptor lists, which are used to specify a list of items as a parameter. You can concatenate or cascade these strings together to create a more complex parameter.

Table 6-1 describes the format of AppleEvent descriptor strings and indicates VIs that can create the descriptor, where appropriate.

Table 54-1. AppleEvent Descriptor String Formats

To send data as:	Format the string as:	Parameter is of code type:	Examples:	VI that can construct string:
an integer	A series of decimal digits, optionally preceded by a minus sign.	long or shor	1234 –5678	n/a
enumerated data	A four-letter code. If it is too long, it is truncated; if it is too short, it is padded with spaces. If you put single quotes (') around it, it can contain any characters; otherwise, it cannot contain: @ ' : - , ([{ }]) and cannot begin with a digit.	enum	whos '@all' long >= '86it'	n/a

Table 54-1. AppleEvent Descriptor String Formats (Continued)

To send data as:	Format the string as:	Parameter is of code type:	Examples:	VI that can construct string:
a string	Enclose the desired sequence of characters within open and close curly quotes (“entered with option-[and” entered with option-shift-[). Notice that the string is not null-terminated.	TEXT	“put x into card field 5” “Hi There”	n/a

an AE record	Enclose a comma-separated list of elements in curly braces, where each element consists of a keyword (a typecode) followed by a colon, followed by a value, which can be any of the types listed in this table.	reco	{x:100, y:–100}
			{'origin': {x:100, y:–100}, extent: {x:500, y:500}, cont:[1,5,25]}
				AECreate Record

an AE descriptor list	Enclose a comma-separated list of descriptors in square brackets.	list	[123, –58, “test”]	AECreate Descriptor List
hex data	Enclose an even number of hex digits between French quotes («entered with option-\ and» entered with option-shift-\).	?? (must be coerced – see next item)	«01 57 64 fe AB C1»	(Hex data is a component of the string produced by Make Alias)

Table 54-1. AppleEvent Descriptor String Formats (Continued)

To send data as:	Format the string as:	Parameter is of code type:	Examples:	VI that can construct string:

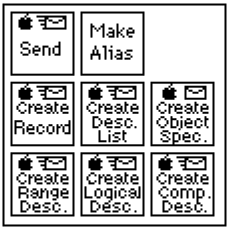
some other data type	Embed data created in one of the types of this table in parentheses and put the desired type code before it. If the data is a numeric, LabVIEW coerces the data to the specified type if possible and returns the `errAECoercionFail` error code if it cannot. If the data is of a different type, LabVIEW replaces the old typecode with the specified type code.	The specified type code	sing(1234)
			alis(«*hex dump of an alias*»)
			type(line)
			rang{star: 5, stop: 6}
				n/a
				Make Alias creates a hex dump of a file description.
				n/a
				n/a

null data	Coerce an empty string to no type.	null	( )	n/a

Low-Level AppleEvent VIs

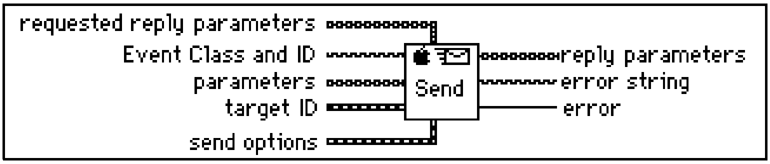
You can use the VIs in this section to construct AppleEvent parameters and send the AppleEvent. The high-level VIs for sending AppleEvents, described earlier in this chapter, are based on the AESend VI, and are good examples of creating AppleEvents and their parameters.

To access the Low Level Apple Events palette, pop up on the Low Level Apple Events icon.



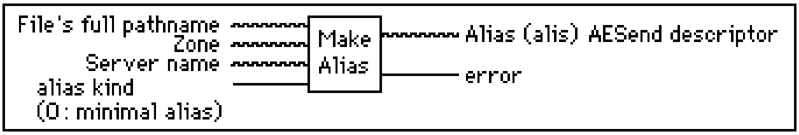
AESend

Sends an AppleEvent specified in parameters to the specified target application.



Make Alias

Creates a unique description of a file from its pathname and location on the network. You can use this description with the AESend VI when sending an AppleEvent that refers to a file.



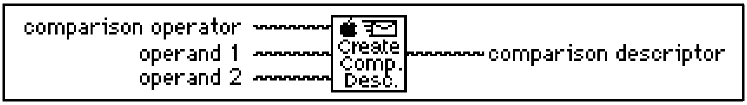
An alias is a data structure used by the Macintosh toolbox to describe file system objects (files, directories and volumes). Do not confuse this with a Finder™ alias file. A minimal alias contains a full path name to the file and possibly the zone and server that the file resides on. A full alias contains more information, such as creation date, file type, and creator. (The complete description of the structure of an alias is confidential to Apple Computer.) Aliases are the most common way to specify a file system object as a parameter to an AppleEvent.

Creating AppleEvent Parameters Using Object Specifiers

Apple has created a high-level interface for creating AppleEvents called the Object Support Library. This interface is actually layered on top of the AppleEvent parameter data structures described earlier in this chapter. This interface helps create common types of parameters, including range specifications. LabVIEW object support VIs are located on the Low Level Apple Events pop up palette.

AECreate Comp Descriptor

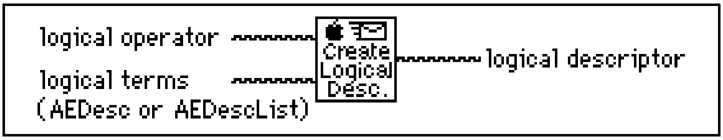
Creates a string describing an AppleEvent comparison record, which specifies how to compare AppleEvent objects with another AppleEvent object or a descriptor record.



For example, you can use the output comparison descriptor string as an argument to the AESend VI, or as an argument to AECreate Object Specifier to build a more complex descriptor string. See the *Object Support VI Example* section of this chapter for an example of its use.

AECreate Logical Descriptor

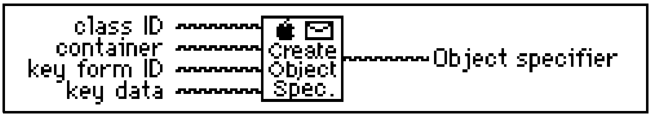
Creates a string describing an AppleEvent logical descriptor, which you use with the AESend VI.



AppleEvent logical records describe logical, or Boolean expressions of multiple terms, such as the AND of two AppleEvent comparison records. For example, you can use the output logical descriptor string as an argument to the AESend VI, or as an argument to AECreate Object Specifier VI to build a more complex descriptor string. See the *Object Support VI Example* in this chapter for an example of its use.

AECreate Object Specifier

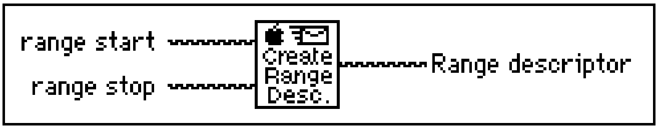
Creates a string describing an AppleEvent object, which you use with the AESend VI.



An object specifier is an AppleEvent record whose type is `obj` and describes a specific object. It has four elements: the class of the object, the containing object, a code indicating the form of the description, and the description of the object.

AECreate Range Descriptor

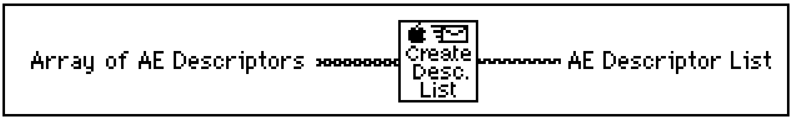
Creates a string describing an AppleEvent range descriptor record, which you use with the AESend VI.



Range descriptor records are used in object specifiers whose key form is formRange (`rang`). They describe a range of objects with two object specifiers: the start and the end of the range

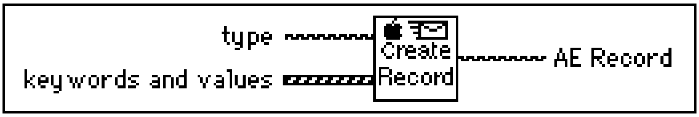
AECreate Descriptor List

Creates a string describing a list of AppleEvent descriptors, which you can then use with the AESend VI. You commonly use Descriptor lists when you create the operands for a logical descriptor



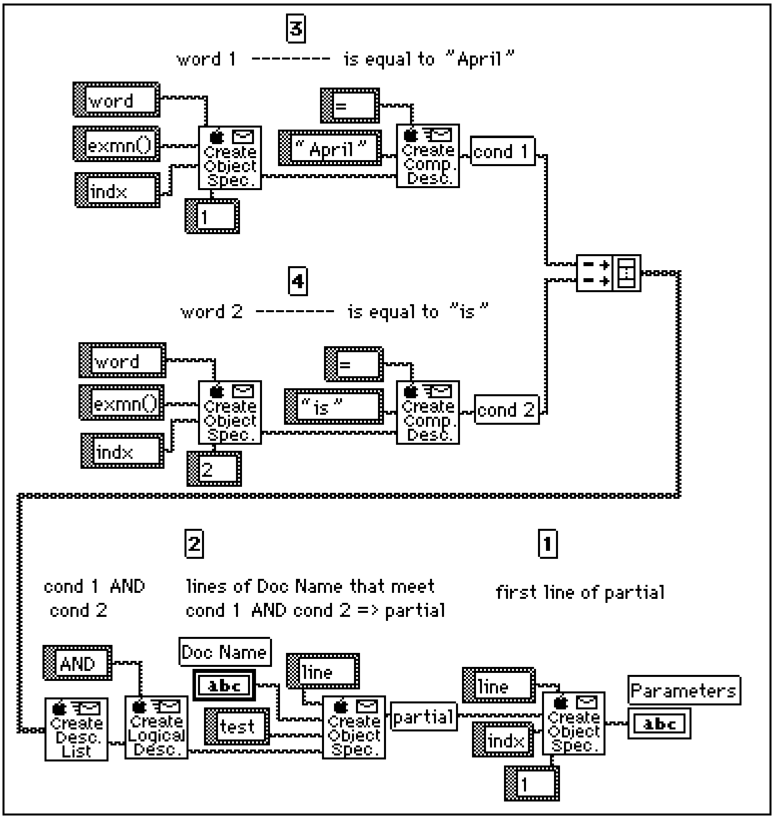
AECreate Record

Creates a string describing an AppleEvent descriptor record, which can then be used with the AESend VI. You can use a record descriptor to bundle descriptors of different types. Each descriptor has its own keyword, or name, and value



Object Support VI Example

The following example creates an AppleEvent parameter using the object support VIs. This example creates an AppleEvent parameter to be sent to a word processor, asking the word processor to return the first line of a specified document whose first word is `April` and whose second word is `is`.



The following string that the previous diagram creates is quite complicated; tabs are added to make the string easier to read. For further information about the Object Support Library consult the *AppleEvent Registry*.

```
obj {
    want: type(‘line’),
    from: obj {
        want: type('line'),
        from: Doc Name,
        form: test,
        seld: logi {
            term:[
          cmpd{
             relo:=,
             obj1:“April”,
             obj2:obj {
                want: type('word'),
                from: exmn( ),
                form: indx,
                seld: 1
                }
             },
          cmpd{
             relo:=,
             obj1:“is”,
             obj2:obj {
                want: type('word'),
                from: exmn( ),
                form: indx,
                seld: 2
                }
             }
          ],
       logc: AND
      }
   },
form: indx,
seld: 1
}
```

Sending AppleEvents to LabVIEW from Other Applications

LabVIEW responds to required AppleEvents, which Apple expects all System 7 applications to support, and to LabVIEW specific

AppleEvents, designed specifically for LabVIEW. Both categories are described in the following sections.

Required AppleEvents

LabVIEW responds to the required AppleEvents, which are Open Application, Open Documents, Print Documents, and Quit Application. These events are described in Inside Macintosh, Volume VI.

LabVIEW Specific AppleEvents

LabVIEW also responds to the LabVIEW specific AppleEvents Run VI, Abort VI, VI Active?, and Close VI. With these events and the Open Documents AppleEvent, you can use other applications to programmatically tell LabVIEW to open a VI, run it, and close it when it is finished. A thorough understanding of AppleEvents, as described in Inside Macintosh, Volume VI, and the AppleEvent Registry is a prerequisite for sending these AppleEvents to LabVIEW from other applications. You can send these events between two or more LabVIEW applications by using the utility VIs described in the Sending AppleEvents section in Chapter 49, *Communication Applications in LabVIEW*.

The LabVIEW specific AppleEvents are described in later sections, in a format similar to that used in the AppleEvent Registry.

Replies to AppleEvents

If LabVIEW is unable to perform an AppleEvent, the reply will contain an error code. If the error is not a standard AppleEvent error, the reply will also contain a string describing the error. The *Error Codes* manual summarizes the LabVIEW specific errors that can be returned in a reply to an AppleEvent.

Event: Run VI

Description

Tells LabVIEW to run the specified VI(s). Before executing this event, the LabVIEW application must be running, and the VI must be open (you can open the VI using the Open Documents AppleEvent).

Event Class

LBVW	(Custom events use the Applications creator type for the event class)

Event ID

GoVI ----

Event Parameters

Description	Keyword	Default Type
VI or List of VIs	`keyDirectObject` (----)	typeChar (char) (required)or list of typeChar (list)

Reply Parameters

Description	Keyword	Default Type
none		

Possible Errors

Error	Value	Description
kLVE_InvalidState	1000	The VI is in a state that does not allow it to run.
kLVE_FPNotOpen	1001	The VI front panel is not open.
kLVE_CtrlErr	1002	The VI has controls on its front panel that are in an error state.
kLVE_VIBad	1003	The VI is broken.
kLVE_NotInMem	1004	The VI is not in memory.

Event: Abort VI

Description

Tells LabVIEW to abort the specified VI(s). Before executing this event, the LabVIEW application must be running, and the VI must be open (you can open the VI using the Open Documents AppleEvent). This message can only be sent to VIs that are executed from the top level (subVIs are aborted only if the calling VI is aborted).

Event Class

LBVW (Custom events use the Applications creator type for the event class)

Event ID

RsVI

Event Parameters

Description	Keyword	Default Type
VI or List of VIs	`keyDirectObject (----)`	typeChar (char) (required)or list of typeChar (list)

Reply Parameters

Description	Required? Keyword	Default Type
none		

Possible Errors

Error	Value	Description
`kLVE_InvalidState`	`1000`	The VI is in a state that does not allow it to run.
`kLVE_FPNotOpen`	`1001`	The VI front panel is not open.
`kLVE_NotInMem`	`1004`	The VI is not in memory.

Event: VI Active?

Description

Requests information on whether a specific VI is currently running. Before executing this event, the LabVIEW application must be running, and the VI must be open (you can open the VI using the Open Documents AppleEvent). The reply indicates whether the VI is currently running.

Event Class

LBVW (Custom events use the Applications creator type for the event class)

Event ID

VIAc

Event Parameters

Description	Keyword	Default Type
VI Name (required)	`keyDirectObject (----)`	typeChar (char)

Reply Parameters

Description	Keyword	Default Type
Active? (required)	`keyDirectObject (----)`	typeBoolean (bool)

Possible Errors.

Error	Value	Description
`kAEvtErrFPNotOpen`	`1001`	The VI front panel is not open.
`kLVE_NotInMem`	`1004`	The VI is not in memory.

Event: Close VI

Description

Tells LabVIEW to close the specified VI(s). Before executing this event, the LabVIEW application must be running, and the VI must be open (you can open the VI using the Open Documents AppleEvent).

Event Class

LBVW (Custom events use the Applications creator type for the event class)

Event ID

ClVI

Event Parameters

Description	Keyword	Default Type
VI or List of VIs	keyDirectObject (----)	typeChar (char) (required)or list of typeChar (list)
Save Options (not required)	keyAESaveOptions (savo)	typeEnum (enum) possible values: yes and no

Reply Parameters

Description	Keyword	Default Type
none		

Possible Errors.

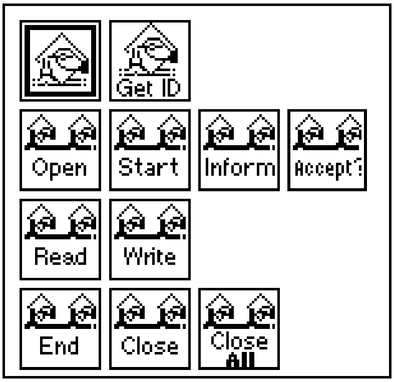
Error	Value	Description
kAEvtErrFPNotOpen	1001	The VI front panel is not open.
kLVE_NotInMem	1004	The VI is not in memory.
cancelError	43	The user cancelled the close operation

Program to Program Communication VIs

Chapter 55

This chapter describes the LabVIEW VIs for program-to-program communication (PPC), a low-level form of Apple interapplication communication (IAC) by which Macintosh applications send and receive blocks of data.

The following illustration shows the **PPC VI** palette, which you access by selecting **Functions»Communication»PPC**.



For examples of how to use the PPC VIs, see the examples located in `examples:comm:PPC Examples.llb`.

Introduction to PPC

PPC is a higher performance protocol than Apple Events because PPC requires less overhead to transmit information. However, because PPC does not define the form or meaning of information that it transfers, it is more complicated to use and many applications do not support it.

LabVIEW VIs can use PPC to send and receive large amounts of information between applications on the same computer or different computers on a network. For two applications to communicate with PPC, they must both be running and prepared to send or receive information. To launch an application remotely, you can use the

AESend Finder Open VI, which is described in the *AppleEvents* section of Chapter 49, *Communication Applications in LabVIEW*.

General PPC Behavior

To communicate using PPC, each application must open a named *port*, over which communication sessions are established, as shown in Figure 55-1. The application that requests communication is the *client*; and the application with which the client communicates is the *server*. The server application makes its availability known by issuing a PPC Inform Session operation. The client requests a session with the server application, which can either accept or reject the request. If the server application accepts the request, then the system establishes a session and the two applications can send and receive blocks of information between them. When the applications finish communicating, you should end the session. You may also want to close the port if you do not want to establish more sessions with that port.

You use the PPC Open Port VI to open a port for communication. PPC Open Port returns a port reference number, which you use in subsequent operations relating to that port. You can have multiple ports open simultaneously, as long as they each have a different name. Each port can support multiple sessions.

You can initiate a session using the PPC Start Session VI. You pass PPC Start Session a **target ID** (see the *General Apple Event VI Behavior* section of Chapter 54, *Apple Event VIs*) and the port reference number through which you want to communicate. If the target application accepts the session, PPC Start Session returns a session reference number, which you use in subsequent communication for that session. PPC Start Session also incorporates an authentication (password) mechanism.

To receive session requests, use the PPC Inform Session VI. You can configure this VI to accept all requests automatically, or you can decide whether to accept or reject the request based on the information about the requesting application that this VI returns. You should accept or reject the request using the PPC Accept Session VI immediately, because the other computer waits (hangs) until you accept or reject its attempt to initiate a session, or until an error occurs.

When a session is established, you can use the PPC Write and PPC Read VIs to communicate with the other application. When you are finished with a session, you should execute the PPC End Session VI and close the port using the PPC Close Port VI.

Figure 55-1 illustrates the order in which you use the PPC VIs.

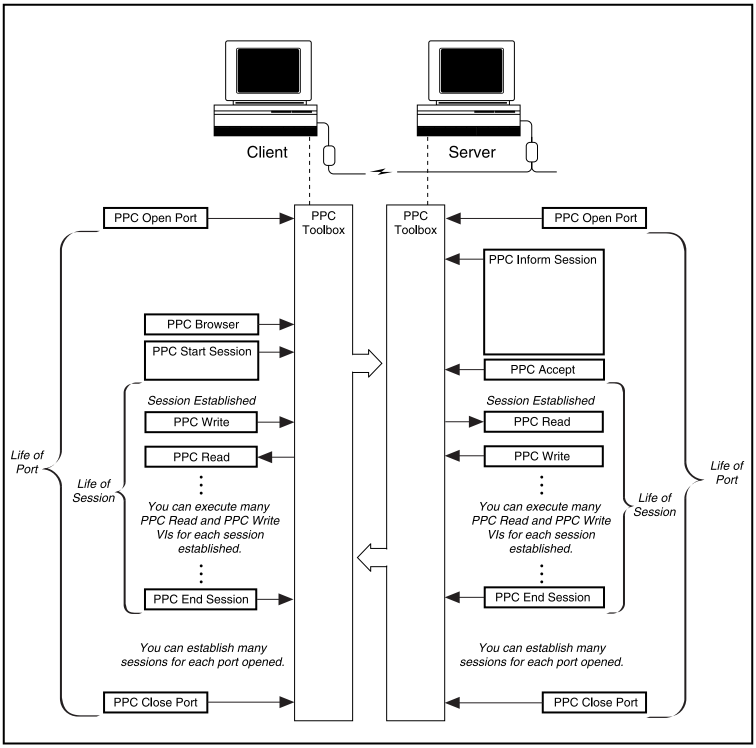


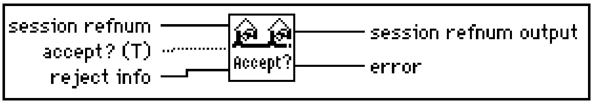
Figure 55-1. PPC VI Execution Order (Used by permission of Apple Computer, Inc.)

PPC VI Descriptions

The following PPC VIs are available.

PPC Accept Session

Accepts or rejects a PPC session request based on the Boolean **accept?**.



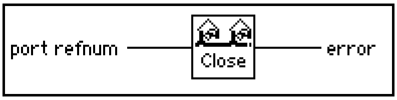
You should accept or reject the request using the PPC Accept Session VI immediately, because the other computer waits (hangs) until the VI accepts or rejects its attempt to initiate a session or an error occurs.

PPC Browser

For information on the PPC Browser VI, see Chapter 54, *Apple Events VIs*, of this manual.

Close All PPC Ports

Closes all the PPC ports that the PPC Open Port VI opened.

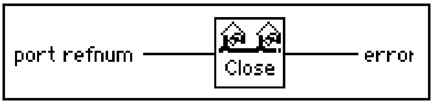


Closing a port terminates all outstanding calls associated with the port with a portClosedErr (error –916).

You can use the Close All PPC Ports to handle abnormal conditions that leave ports open. An example of an abnormal condition is when a VI is aborted before it can terminate normally and close the PPC port. You can use the Close All PPC Ports VI during VI development, when such mistakes are more likely to be made, or as a precaution at the beginning of any program that opens ports.

PPC Close Port

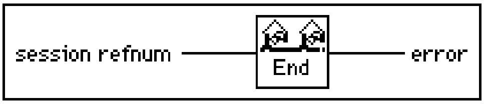
Closes the specified PPC port.



Closing a port terminates all outstanding calls associated with the port with a portClosedErr (error –916).

PPC End Session

Ends the specified PPC session.



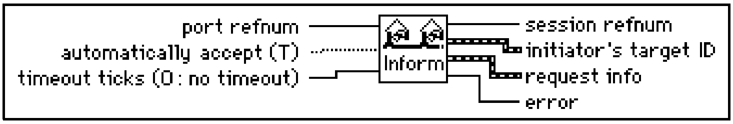
Ending a session causes all outstanding calls associated with the session (PPC Read and PPC Write calls) to finish with a sessClosedErr (error -917).

Get Target ID

For information on the Get Target ID VI, see Chapter 54, *Apple Event VIs*, of this manual.

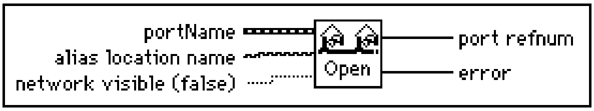
PPC Inform Session

Waits for a PPC session request.

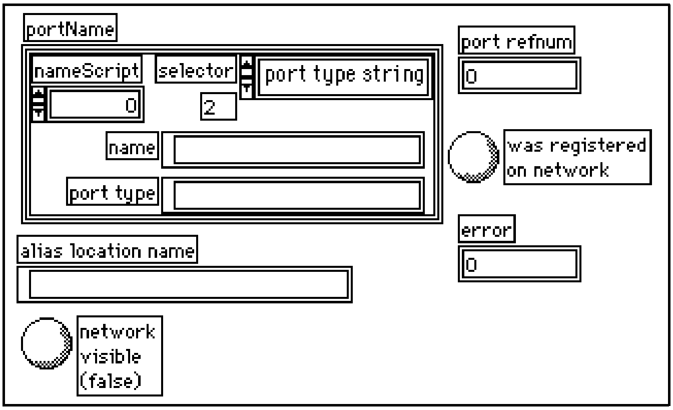


PPC Open Port

Opens a port for PPC communication and returns a unique port reference number in **port refnum**. You can use a single port for multiple sessions.



When opening a port using PPC Open Port, you must specify a **portName** cluster.

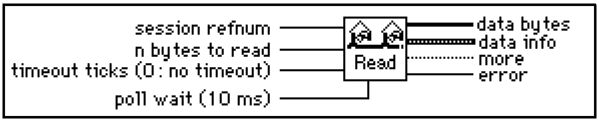


nameScript 0
selector 2
port refnum 0
error 0

Refer to the LabVIEW online help for more information on this VI.

PPC Read

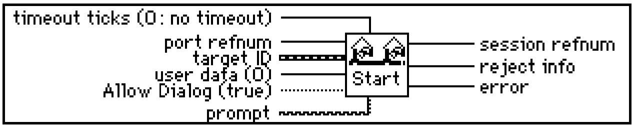
Reads a block of information from a specified session. If a timeout occurs or the VI aborts before completing execution, the port that **port refnum** represents closes.



PPC Read executes asynchronously by starting to read the specified data and then polling until the read is finished.

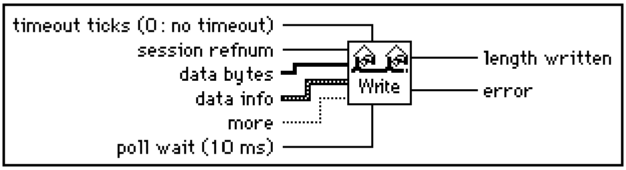
PPC Start Session

Attempts to start a session with the application specified by **target ID** through the specified port. If a timeout occurs or the VI aborts before completing execution, the port represented by **port refnum** closes.



PPC Write

Writes a block of information to the specified session. If a timeout occurs or the VI aborts before completing execution, the port represented by **port refnum** is closed. PPC Write executes asynchronously by starting to write the specified data and then polling until the write is finished.



Appendix A

DAQ Hardware Capabilities

This appendix contains tables that summarize the analog and digital I/O capabilities of National Instruments data acquisition (DAQ) devices. The devices in this appendix are grouped into categories. The DAQ device categories for these tables include the following.

- MIO and AI Devices
- Lab and 1200 Series and Portable Devices
- 54xx Series Devices
- SCXI Modules
- Dynamic Signal Acquisition Devices
- Analog Output Only Devices
- Digital Only Devices
- Timing Only Devices
- 5102 Devices Hardware Capabilities

Note: ***(Macintosh) When a NuBus device indicates it supports DMA transfers, a DMA device (such as an NB-DMA2800) is also required.***

MIO and AI Device Hardware Capabilities

Table A-1. Analog Input Configuration Programmability—MIO and AI Devices

Device	Gain	Range	Polarity	SE/DIFF	Coupling
All MIO-E Series Devices All AI-E Series Devices	By Channel	By Channel	By Channel	By Channel	DC
AT-MIO-16F-5	By Channel	By Group	By Group	By Group	DC
AT-MIO-64F-5 AT-MIO-16X	By Channel	By Channel	By Channel	By Channel	DC
AT-MIO-16/16D NB-MIO-16 NB-MIO-16X	By Channel	By Device	By Device	By Device	DC

"By device" means you select the value of a parameter with hardware jumpers, and the selection affects any group of channels on the device. "By group" means you program the selection through software, and the selection affects all the channels used at the same time. "By channel" means you program the selection with hardware jumpers or through software on a per channel basis. When a specific value for a parameter is shown, that parameter value is fixed.

Table A-2. Analog Input Characteristics—MIO and AI Devices (Part 1)

Device	Number of Channels	Resolution	Gains[1]	Range (V)[1]	Input FIFO (words)	Scanning[2]
AT-MIO-16E-1 AT-MIO-16E-2 AT-MIO-16E-10 AT-MIO-16DE-10 NEC-MIO-16E-4 PCI-MIO-16E-1 PCI-MIO-16E-4 NEC-AI-16E-4	16SE, 8DI	12 bits	0.5, 1, 2, 5, 10, 20, 50, 100	±5, 0 to 10	512; E-1: 8,192; E-2 and E4: 2,048	Up to 512
AT-MIO-64E-3*	64SE, 32DI	12 bits	0.5, 1, 2, 5, 10, 20, 50, 100	±5, 0 to 10	2,048	Up to 512
PCI-MIO-16XE-10	16SE, 8DI	16 bits	1, 2, 5, 10, 20, 50,100	±10, 0 to 10	512	Up to 512

Table A-2. Analog Input Characteristics—MIO and AI Devices (Part 1) (Continued)

Device	Number of Channels	Resolution	Gains[1]	Range (V)[1]	Input FIFO (words)	Scanning[2]
NEC-MIO-16XE-50 NEC-AI-16XE-50 AT-MIO-16XE-50 DAQPad-MIO-16XE-50 PCI-MIO-16XE-50	16SE, 8DI	16 bits	1, 2,10, 100	±10, 0 to 10	512	Up to 512
AT-MIO-16F-5 AT-MIO-64F-5**	16SE, 8DI 64SE, 32DI	12 bits	0.5, 1, 2, 5, 10, 20, 50, 100	±5, ±10, 0 to 10	16F-5: 256; 64F-5: 512	Up to 512
AT-MIO-16X	16SE, 8DI	16	1, 2, 5, 10, 20, 50, 100	±10, 0 to 10	512	Up to 512

AT-MIO-16(L)
AT-MIO-16(H)
AT-MIO-16D(L)
AT-MIO-16D(H)
	16SE, 8DI	12	(L) 1, 10, 100, 500; (H): 1, 2, 4, 8	±5, ±10, 0 to 10	16 (L,H);
					512 (DL, DH)
						Up to 16

NB-MIO-16 NB-MIO-16X	16SE, 8DI	MIO-16: 12; MIO-16X: 16	(L) 1, 10, 100, 500; (H) 1, 2, 4, 8	±10, ±5, 0 to 10, 0 to 5	16; MIO-16, Rev. G: 512	Up to 16; MIO-16: groups of 2, 4, 8, and 16

[1] You can determine the limit settings of your device by multiplying the range and the voltage values together. For more information on limit settings in LabVIEW, refer to the Basics LabVIEW Data Acquisition Concepts chapter in the LabVIEW Data Acquisition Basics Manual.

[2] Scanning = channels, in any order.

*The valid channels for the AT-MIO-64E-3 in Differential Mode are 0-7, 16-23, 32-39, and 48-55.

**The valid channels for the AT-MIO-64F-5 in Differential Mode are 0-7 and 16-39.

Table A-3. Analog Input Characteristics—MIO and AI Devices (Part 2)

Device	Triggers[1]	Max Sampling Rate (S/s)	Transfer Method
AT-MIO-16E-1 AT-MIO-16E-2 AT-MIO-64E-3 AT-MIO-16E-10 AT-MIO-16DE-10 PCI-MIO-16E-1 PCI-MIO-16XE-10 NEC-AI-16E-4 NEC-MIO-16E-4 PCI-MIO-16E-4	SW, Pre, Post, (and Analog on E-1, E-2, E-3, and E-4 only)	E-1: 1 M, E-2 and E-3: 500 k, E-4: 250 k, E-10 and DE-10: 100 k	DMA, interrupts
All MIO-16XE-50 Devices NEC-AI-16XE-50	SW, Pre, Post	20 k	DMA, (interrupts on DAQPad-MIO-16XE-50)
AT-MIO-16F-5 AT-MIO-64F-5	SW, Pre, Post	200 k	DMA, interrupts
AT-MIO-16X AT-MIO-16/16D	SW, Pre, Post	100 k	DMA, interrupts
NB-MIO-16	SW, Post	111 k (L-9 or H-9), 67 k (L-15 or H-15), 40 k (L-25 or H-25)	DMA, interrupts
NB-MIO-16X	SW, Post	55 k (L-18 or H-18), 24 k (L-42 or H-42)	DMA, interrupts

[1] SW=Software Triggering (also called conditional retrieval), Pre=Pretrigger, Post=Posttrigger.

☞ **Note:** ***For NB-MIO devices, software triggering is actually done in the interrupt service routine (interrupts only) and is different than conditional retrieval.***

Table A-4. Analog Output Characteristics—MIO and AI Devices

Device	Channel Numbers	DAC Type	Output Limits	Update Clocks	Waveform Grouping	Transfer Method
All MIO-16E Devices AT-MIO-16DE-10 AT-MIO-64 E-3 AT-MIO-16XE-50 DAQPad-MIO-16XE-50 PCI-MIO-16E-1 PCI-MIO-16E-4 PCI-MIO-16XE-50	0, 1	12-bit double buffered (E-1, E-2, 64E-3, and E-4: 2 K FIFO)	0 to 10, ±10, ±Vref, 0 to Vref (only ±10 on XE-50 devices)	Update clock 1 or external update.	0, 1, or 0 and 1	DMA, interrupts
PCI-MIO-16XE-10		16-bit	±10, 0 to 10			
AT-MIO-16F-5 AT-MIO-64F-5	0, 1	12-bit double buffered (64F-5: 2 K FIFO)	0 to 10, ±10, ±Vref, 0 to Vref	Update clock 1 is first available of ctr 5, 2, 1 or external update. Default is 5. Timebase signal range is 5,000,000, 1,000,000, 100,000, 10,000, 1,000, and 100.	0, 1, or 0 and 1	DMA, interrupts
AT-MIO-16X	0, 1	16-bit double buffered (2 K FIFO)	±10, 0 to 10, ±Vref, 0 to Vref	Update clock 1 is first available on ctr 5, 2, 1, or external update. Timebase signal range is 5,000,000, 1,000,000, 100,000, 10,000, 1,000, 100.	0, 1, or 0 and 1	DMA, interrupts

Table A-4. Analog Output Characteristics—MIO and AI Devices (Continued)

Device	Channel Numbers	DAC Type	Output Limits	Update Clocks	Waveform Grouping	Transfer Method
AT-MIO-16/16D	0, 1	12-bit double buffered	0 to 10, ±10, ±Vref, 0 to Vref	Update clock 1 is ctr2 or external update. Timebase signal range is 1,000,000, 100,000, 10,000, 1,000, and 100.	0, 1, or 0 and 1	Interrupts
NB-MIO-16/16X	0, 1	MIO-16: 12-bit ; MIO-16X: 12-bit double buffered	0 to 10, ±10, ±Vref, 0 to Vref	Update clock 1, external update (MIO-16X only). Timebase signal range is 1,000,000, 100,000, 10,000, 1,000, and 100.	0, 1, or 0 and 1	MIO-16: DMA; MIO-16X: DMA, interrupts

Table A-5. Digital I/O Hardware Capabilities—MIO and AI Devices

Device	Port Type	Port Numbers	Handshake Modes	Direction	DIO Clocks	Transfer Method
All MIO-16 Devices AT-MIO-16D[1] AT-MIO-64F-5	4-bit ports	0, 1	No handshaking	Read or write	None	Software polling
All MIO-16E Devices All NEC-E Series Devices AT-MIO-64E-3 AT-MIO-16DE-10[1] AT-MIO-16XE-50 DAQPad-MIO-16XE-50 PCI-MIO-16XE-50	8-bit ports	0	No handshaking	Bit-wise direction control	None	Software polling
AT-MIO-16D[1] AT-MIO-16DE-10[1]	8-bit ports	2, 3	Handshaking on or off	Read or write, port 2 may be bi-directional	None	Interrupts
	8-bit ports	4	No handshaking; Unusable if port 2 or 3 uses handshaking	Read or write	None	Software polling

[1] These devices appear more than once in this table, because they have enhanced digital functionality.

Table A-6. Counter Characteristics—MIO and AI Devices

Device	Counter Chip Used	# of General Purpose Counters Available	Timebases Available	Number of Bits	Gate Modes Available	Out-puts Available	Output Modes Available	Count Direction[1]
E Series Devices	DAQ-STC	2	2 internal: 20 MHz or 100 kHz; external	24	rising-edge, falling-edge, high-level, low-level	2		up or down, can be SW- or HW-controlled
AT-MIO-16F-5 AT-MIO-64F-5 AT-MIO-16/16D NB-MIO-16/16X	Am-9513	3	5 or 6 internal: 5 MHz (only on CTR2 of 16F-5, 64F-5, and AT-MIO-16X), 1 MHz, 100 kHz, 10 kHz, 1 kHz, 100 Hz; external	16	rising-edge, falling-edge, high-level, low-level	2	TC pulse or TC toggle	Up

[1] SW = Software; HW = Hardware.

Table A-7. Counter Usage for Analog Input and Output—MIO and AI Devices

Device name	Counter Chip Used	AI Channel Clock	AI Sample Counter	AI Scan Clock	AO Update Clock
E Series Devices	DAQ-STC	The DAQ-STC chip uses dedicated clocks for these purposes.			
AT-MIO-16F-5 AT-MIO-64F-5 AT-MIO-16X	Am9513	Ctr 3	Ctr 4 (& 5)[1]	Ctr 2 (or 1)[2]	Ctr 5, 2 or 1
AT-MIO-16/16D NB-MIO-16X	Am9513	Ctr 3	Ctr 4 (& 5)[1]	Ctr 2 (or 1)[2]	Ctr 2 (and via DMA for NB-MIO-16X)
NB-MIO-16	Am9513	Ctr 3	Ctr 4 (& 5)[1]	None (or 1)[2]	(via DMA)

[1] If the total number of samples is less than 65535, only the first counter is used. If the number of samples exceeds 65536, the first counter is used together with the second counter as a 32-bit sample counter.

[2] Ctr 2 (or no counter for NB-MIO-16) is used for normal scanning operations, and Ctr 1 is used for AMUX-64T and SCXI hardware scanning.

Lab and 1200 Series and Portable Devices Hardware Capabilities

Table A-8. Analog Input Configuration Programmability—Lab and 1200 Series and Portable Devices

Device	Gain	Range	Polarity	SE/DIFF	Coupling
Lab-LC Lab-NB	By group	By device	By device	SE	DC
Lab-PC+	By group	By group	By device	By device	DC
SCXI-1200 DAQPad-1200 DAQCard-1200 PCI-1200	By group	By group	By group	By group	DC
DAQCard-500	1	Only 1 range available	Bipolar	SE	DC
DAQCard-516/PC-516	1	Only 1 range available	Bipolar	By group	DC
DAQCard-700	1	By group	Bipolar	By group	DC
PC-LPM-16	1	By device	Bipolar	SE	DC

☞ **Note:** ***"By device" means you select the value of a parameter with hardware jumpers, and the selection affects any group of channels on the device. "By group" means you program the selection through software, and the selection affects all the channels used at the same time. "By channel" means you program the selection with hardware jumpers or through software on a per channel basis. When a specific value for a parameter is shown, that parameter value is fixed.***

Table A-9. Analog Input Characteristics—Lab and 1200 Series and Portable Devices (Part 1)

Device	Number of Channels	Resolution (bits)	Gains[1]	Range (V)[1]	Input FIFO (samples)
Lab-LC Lab-NB	8SE	12	1, 2, 5, 10, 20, 50, 100	±5, 0 to 10	16
Lab-PC+ SCXI-1200 DAQPad-1200 DAQCard-1200 PCI-1200	8SE, 4DI	12	1, 2, 5, 10 20, 50, 100	±5, 0 to 10	2,048; Lab-PC: 512
DAQCard-500	8SE	12	1	±5	16
DAQCard 516 PC516	8SE,4DI	16	1	+/-5	512
DAQCard-700	16SE, 8DI	12	1	±10, ±5, ±2.5	512
PC-LPM-16	16SE	12	1	±5, ±2.5, 0 to 10, 0 to 5	16

[1] You can determine the limit settings of your device by multiplying the range and the voltage values together. For more information on limit settings in LabVIEW, refer to the Basics LabVIEW Data Acquisition Concepts chapter in the LabVIEW Data Acquisition Basics Manual.

Table A-10. Analog Input Characteristics—Lab and 1200 Series and Portable Devices (Part 2)

Device	Scanning	Triggers	Max Sampling Rate (S/s)	Transfer Method
Lab-LC Lab-NB	Any single channel; for multiple channels, *N* through 0, where *N*<=7	Software trigger, pretrigger, and posttrigger with digital trigger	62.5 k	Interrupts
Lab-PC+ SCXI-1200 DAQPad-1200 DAQCard-1200	Any single channel; for multiple channels, N through 0, where N<=7.	Software trigger, pretrigger, and posttrigger with digital trigger	100 k; Lab-PC+: 83 k	Interrupts; Lab-PC+: Interrupts, DMA
DAQCard-500 DAQCard 516 PC-516	Any single channel; for multiple channels, N through 0, where N<=7	Software trigger only	50 k	Interrupts

Table A-10. Analog Input Characteristics—Lab and 1200 Series and Portable Devices (Part 2) (Continued)

Device	Scanning	Triggers	Max Sampling Rate (S/s)	Transfer Method
DAQCard-700	Any single channel; for multiple channels, N through 0, where N#15	Software trigger only	100 k	Interrupts
PC-LPM-16	Any single channel; for multiple channels, N through 0, where N#15	Software trigger only	50 k	Interrupts

Table A-11. Analog Output Characteristics—Lab and 1200 Series and Portable Devices

Device	Channel #s	DAC Type	Output Limits (V)	Update Clocks	Waveform Grouping	Transfer Methods
Lab-NB Lab-LC	0, 1	12-bit double-buffered	0 to 10, ±5	Update clock 1 is ctrA2 or external update; timebase is 1 MHz or ctrB0	0, 1, or 0 and 1	Interrupts
Lab-PC+ SCXI-1200 DAQPad-1200 DAQCard-1200 PCI-1200	0, 1	12-bit double-buffered	0 to 10, ±5	Update clock 1 is ctrA2 or external update; timebase signal range is 1,000,000, 100,000, 10,000, 1,000, and 100	0, 1, or 0 and 1	Interrupts

☞ **Note:** ***The DAQCard-516 and PC 516 devices do not have analog output.***

Table A-12. Counter Usage for Analog Input and Output—Lab Series and Portable Devices

Device Name	Counter Chip Used	AI Channel Clock	AI Sample Counter	AI Scan Clock	AO Update Clock
Lab-NB, Lab-LC	82C53	Ctr A0 (& B0)[1]	Ctr A1	None	Ctr A2
Lab-PC+, DAQPad-1200, SCXI-1200, DAQCard-1200, PCI-1200	82C53	Ctr A0 (& B0)[1]	Ctr A1	Ctr B1	Ctr A2
DAQCard-500, DAQCard-700,	8254	Ctr 0	(software)	None	None
DAQCard 516 PC-516	82C54	Ctr0	SW	None	None
PC-LPM-16	82C53	Ctr 0	(software)	None	None

[1] The second counter is used as an extended timebase for timed analog input or output when sample interval exceeds 65.535 ms.

Table A-13. Digital I/O Hardware Capabilities—Lab and 1200 Series and Portable Devices

Device	Port Type	Port Numbers	Handshake Modes	Direction	DIO Clocks	Transfer Method
Lab-NB Lab-LC Lab-PC+ SCXI-1200 DAQCard-1200 DAQPad-1200 PCI-1200	8-bit port	0, 1	Handshaking on or off	Read or write, port 0 may be bidirectional	None	Interrupts
	8-bit port	2	No handshaking; unusable if port 0 or 1 uses handshaking	Read or write	None	Software polling
PC-LPM-16	8-bit ports	0, 1	No handshaking	0: read or write	None	Software polling
DAQCard-500	4-bit ports	0, 1	No handshaking	0: write, 1: read	None	Software polling
DAQCard-700	8-bit ports	0, 1	No handshaking	0: write, 1: read	None	Software polling

54xx Devices

Table A-14. Analog Output and Digital Output Characteristics—54XX Series Devices

Characteristics	AT-5411, PCI-5411
Channel Numbers	0
Maximum Update Rate	40 MHz.
Update Interval	1 to 65535.
DAC Type	12-bit, double buffered.
Output Limits (V) (Internal reference only)	±5 into 50 Ω load ±10 into unterminated (high input impedance) load.
Update Clocks	Update clock 1.
Triggers	On rising TTL edge, at trigger input connector or RTSI pin. Can be also generated internally by software.
RTSI Trigger Bus	Yes
Digital Outputs	16-bits with clock signal
Waveform Grouping	0
Waveform Memory Depth -ARB Mode -Direct Digital Synthesis (DDS) Mode	 2,000,000 16-bit samples (standard) 16,384 16-bit samples maximum
Maximum Waveform Stages	290
Buffer Numbers	1 to 1,000.
Buffer Iterations	1 to 65,535
Buffer Sample Count -ARB Mode	256 samples minimum Memory depth maximum Note: Buffer size should be a multiple of 8 samples.
- DDS Mode	Must be equal to 16,384 samples. If you load less number of samples then you will see the contents of unfilled sections of memory also appearing in the waveform generation.
Marker Output	TTL level, One available for every stage

Table A-14. Analog Output and Digital Output Characteristics—54XX Series Devices (Continued)

Characteristics	AT-5411, PCI-5411
DDS Accumulator Size	32-bit
Maximum Output Frequency	16 MHz
Output Frequency Resolution (DDS Mode only)	9.31 mHz
Output Attenuation (after the DAC)	0 through 74.000 dB (Decibels) in 0.001 dB steps
SYNC Output Duty Cycle (% High)	TTL level, 20% to 80%.
PLL Reference Clock	1 MHz, 10 MHz or 20 MHz
Output Enable	software switchable to ON or OFF
Output Impedance	50Ω or 75Ω (video), software selectable
Low-Pass Filter	16 MHz, software switchable to ON or OFF
Digital Half-Band Interpolating Filter	80 MSPS, software switchable to ON or OFF
Trigger Operation Modes	Single, Continuous, Stepped and Burst

☞ **Note:** ***Refer to your hardware user reference manual for default settings of your device.***

Table A-15. Counter/Timer Characteristics -- Lab and 1200 Series and Portable Devices

Device	Counter Chip Used	# of Gen. Purpose Counters Available	Timebases Available	Number of Bits	Gate Modes Available	Outputs Available	Output Modes Available	Count Direction
Lab-NB Lab-LC Lab-PC+ SCXI-1200 DAQCard-1200 DAQPad-1200 PC-LPM-16 PCI-1200	8253	3 (2 with SOURCE input at I/O Connector)	Internal: 1 MHz; (PC-LPM-16: only on CTRB0) external	16	high-level or rising-edge depending on output mode	3	Refer to ICTRControl VI description on modes in Chapter 19, Advanced Counter VIs.	down
DAQCard-500 DAQCard 516 DAQCard-700 PC-516	8254	3 (2 with SOURCE input at I/O Connector)	Internal: 1 MHz only on CTRB0; external	16	high-level or rising-edge depending on output mode	3 (2 for DAQCard -500)	Refer to ICTRControl VI description on modes in Chapter 19, Advanced Counter VIs.	down

SCXI Module Hardware Capabilities

Table A-16. Analog Input Characteristics—SCXI Modules (Part 1)

Module	Number of Channels	Input Voltage Range (V)	Gains[1]	Filter[1]	Excitation Channels[1]	Mode Support
SCXI-1100	32 DI	±10	1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000 (SW/M)[1]	lowpass filter (or no filter) with 10 kHz or 4 Hz cutoff frequency (JS/M)[1]	—	multiplexed
SCXI-1102	32 DI	±10	1, 100 (SW/C)[1]	1 Hz lowpass on each channel	—	multiplexed

Table A-16. Analog Input Characteristics—SCXI Modules (Part 1) (Continued)

Module	Number of Channels	Input Voltage Range (V)	Gains[1]	Filter[1]	Excitation Channels[1]	Mode Support
SCXI-1120 SCXI-1121	8 DI (SCXI-1120) 4 DI (SCXI-1121)	±5	1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, and 2,000 (JS/C)[1]	lowpass filter with 10 kHz or 4 Hz cutoff frequency (JS/C)[1]	SCXI-1121 only: 4 voltage or current excitation JS/C [1] (channels)	multiplexed or parallel
SCXI-1120D	8 DI (SCXI-1120) 4 DI (SCXI-1121)	±5	0.5, 1, 2.5, 5, 10, 25, 50, 100, 250, 500, 1,000	4,500, 24,500 Hz	SCXI-1121 only: 4 voltage or current excitation JS/C [1] (channels)	multiplexed or parallel
SCXI-1122	16 DI or 8 DI and 8 excitation SW/M1 channels	±10	0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 2,000 (SW/M)[1]	lowpass filter with 4kHz or 4 Hz cutoff frequency	8 voltage or current excitation channels in 4-wire scanning mode	multiplexed
SCXI-1140	8 DI, sample and hold	±10	1, 10, 100, 200, 500 (DS/C)[1]	none	—	multiplexed or parallel
SCXI-1141	8 DI	±5	1, 2, 5, 10, 20, 50, 100 (SW/C)[1]	elliptic lowpass filter with 10Hz to 25KHz cutoff frequency[2] (SW/M)[1] (disabled on a per channel basis)	—	multiplexed or parallel

[1] DS/C = dip switch-selectable per channel, JS/C = jumper-selectable per channel, JS/M = jumper-selectable per module, SW/C = software-selectable per channel, SW/M = software-selectable per module

[2] The SCXI-1141 has an automatic filter setting. LabVIEW sets the filter frequency based on the scan rates used with the module.

Table A-17. Analog Output Characteristics—SCXI Modules

Module	Number of Channels	Output Voltage Range (V or mA)	Mode Support
SCXI-1124	6 voltage or current	0 to1, 0 to 5, 0 to 10, ±1, ±5, ±10 (software-selectable) or 0 to 0.20 mA	multiplexed

Table A-18. Relay Characteristics—SCXI Modules

Module	Number of Channels[1]	Latched or Non-latched	Start-up Relay Position[1]	Mode Support
SCXI-1160	16	Latched	Leave relays in the position at power-down.	multiplexed
SCXI-1161	8	Non-latched	Switch to the Normally Closed (NC) position—when the hardware reset is set on the module.	multiplexed

[1] You can set or reset each SCXI relay individually without affecting other relays, or you can change all of the relays at once.

Table A-19. Digital Input and Output Characteristics—SCXI Modules

Module	Type of Module	Number of Channels[1]	Input Voltage Range	Mode Support
SCXI-1162	Input	32 (optically-isolated)	0 to 5 V	Parallel support—when connected to a DIO-24, DIO-96, or DIO-32F device. Multiplexed support with any DAQ device supporting SCXI.
SCXI-1162HV	Input	32 (optically-isolated)	AC or DC signals up to ±240 V	Parallel support—when connected to a DIO-24, DIO-96, or DIO-32F device. Multiplexed support with any DAQ device supporting SCXI.

Table A-19. Digital Input and Output Characteristics—SCXI Modules (Continued)

Module	Type of Module	Number of Channels[1]	Input Voltage Range	Mode Support
SCXI-1163	Output	32 (optically-isolated)	0 to 5 V	Parallel support—when connected to a DIO-24, DIO-96, or DIO-32F device. Multiplexed support with any DAQ device supporting SCXI.
SCXI-1163R*	Output	32 (optically-isolated)	±240 V	Parallel support—when connected to a DIO-24, DIO-96, or DIO-32F device. Multiplexed support with any DAQ device supporting SCXI.

[1] Functionally equivalent to the SCXI-1163, but incorporates solid-state relays in place of digital outputs.

Table A-20. Terminal Block Selection Guide—SCXI Modules

SCXI Module	Terminal Blocks	Cold-Junction Compensation Sensor (CJC)
SCXI-1100 SCXI-1102	SCXI-1303 SCXI-1300	Thermistor IC Sensor
SCXI-1120 SCXI-1121	SCXI-1320 SCXI-1321[1] SCXI-1327 SCXI-1328	IC Sensor IC Sensor Thermistor Thermistor
SCXI-1122	SCXI-1322	Thermistor
SCXI-1124	SCXI-1325	—
SCXI-1140	SCXI-1301 SCXI-1304	— —
SCXI-1141	SCXI-1304	—
SCXI-1160	SCXI-1324	—
SCXI-1161	None–screw terminals located in module.	—

Table A-20. Terminal Block Selection Guide—SCXI Modules (Continued)

SCXI Module	Terminal Blocks	Cold-Junction Compensation Sensor (CJC)
SCXI-1162 SCXI-1162HV SCXI-1163 SCXI-1163R	SCXI-1326	—
SCXI-1180	SCXI-1302	—
SCXI-1181	SCXI-1300 SCXI-1301	IC Sensor —
SCXI-1200	SCXI-1302 CB-50	— —

[1] SCXI-1121 only

Table A-21. Analog Input Configuration Programmability

Device	Gain	Coupling
5102 devices	by channel	by channel

Table A-22. Analog Input Configuration Programmability

Device	Number of Channels	Resolution	Gains	Range (V)	Input FIFO (words)	Scanning
5102 devices	2	8 bits	1, 5, 20, 100	+/- 5	663546	1 or 2 channels in any order without repetitions

Note: ***"By device" means you select the value of a parameter with hardware jumpers, and the selection affects any group of channels on the device. "By group" means you program the selection through software, and the selection affects all the channels used at the same time. "By channel" means you program the selection with hardware jumpers or through software on a per channel basis. When a specific value for a parameter is shown, that parameter value is fixed.***

Analog Output Only Devices Hardware Capabilities

Table A-23. Analog Output Characteristics--Analog Output Only Devices

Device	Channel #s	DAC Type	Output Limits (V)	Update Clocks	Waveform Grouping	Transfer Method
AT-AO-6 AT-AO-10 NB-AO-6	0 through 5, 6 through 9*	12-bit double-buffered with 1 K FIFO for update clock 1 channels	±10, ±Vref1, 0 to 10, 0 to Vref1, 4 to 20 mA, 4 to	Update clock 1 is ctr0 or external update. Update clock 1 channels are 0, 1, 2, 3, 4, 5, 6*, 7*, 8*, 9*, 0 to 1, 0 to 3, 0 to 5, 0 to 7*, 0 to 9*. Update clock 2 is ctr1. Update clock 2 channels are 2, 3, 4, 5, 6*, 7*, 8*, 9*, 2 to 3, 2 to 5, 2 to 7*, 2 to 9*; timebase signal range is 1,000,000, 100,000, 10,000, 1,000, 100	For update clock 1 channels are any one channel *N* or set of channel pairs: 0-*N;* for update clock 2 channels are 2-*N*, same rules as above: *N*#6, *N*#10*	Update clock 1 channels: DMA, interrupts; update clock 2 channels: interrupts
PC-AO-2DC (Plug and Play)	0, 1		0 to 10, ±5, 0-20mA sink software-selectable			
DAQCard-AO-2DC	0, 1		0 to 10, ±5, 0-10mA sink software-selectable			

*AT-AO-10 only

Dynamic Signal Acquisition Devices Hardware Capabilities

Table A-24. Analog Input Configuration Programmability—Dynamic Signal Acquisition Devices

Device	Gain	Range (V)	Polarity	SE/DIFF	Coupling
EISA-A2000 NB-A2000	1	±5	Bipolar	SE	By channel
NB-A2100 AT-DSP2200	1	±2.828	Bipolar	SE	By group
NB-A2150 AT-A2150	1	±2.828	Bipolar	SE	By channel pair 0 and 1, 2 and 3

☞ **Note:** ***"By device" means you select the value of a parameter with hardware jumpers, and the selection affects any group of channels on the device. "By group" means you program the selection through software, and the selection affects all the channels used at the same time. "By channel" means you program the selection with hardware jumpers or through software on a per channel basis. When a specific value for a parameter is shown, that parameter value is fixed.***

Table A-25. Analog Input Characteristics—Dynamic Signal Acquisition Devices

Device	Number of Channels	Resolution	Range (V)	Input FIFO (words)	Triggers	Scanning	Max Sampling Rate (S/s)	Transfer Method
EISA-A2000 NB-A2000	4 SE	12 bits	±5	EISA: 512; NB: 1,024	Software trigger, pretrigger, and posttrigger with digital or analog triggering and posttrigger delay	0, 1, 2, 3, 0 and 1, 2 and 3, 0 to 3.	1M	DMA, interrupts
NB-A2100 NB-A2150	2 SE	16 bits	±2.828	32	Software trigger, pretrigger, and posttrigger with digital or analog triggering	A2150: 0, 1, 2, 3, 0 and 1, 2 and 3, 0 to 3; A2100: 0, 1, 0 and 1	2100: 48 k, 2150: 24 k, 2150C: 48 k, 2150S: 51.2 k	DMA, interrupts
AT-A2150	4 SE	16 bits	±2.828	—	Software trigger, pretrigger, and posttrigger with digital or analog triggering	0, 1, 2, 3, 0 and 1, 2 and 3, 0 and 3	2150: 24 k 2150: 51.2 k	DMA, interrupts

Digital Only Devices Hardware Capabilities

Table A-26. Digital Hardware Capabilities—Digital I/O Devices

Device	Port Type	Port #s	Handshake Modes	Direction	DIO Clocks	Transfer Method
AT-DIO-32F NB-DIO-32F	8-bit ports	0, 1, 2, 3	8-bit port Handshaking on or off; extensive handshaking modes	Read or write	Two clocks available 16-bit with variable timebase	DMA for each group; dual channel DMA for groups containing port 0
	2-bit ports	4	No handshaking	Read or write	None	Software polling
PC-DIO-24 NB-DIO-24 DAQCard-DIO-24	8-bit port	0, 1	Handshaking on or off	Read or write, port 0 may be bidirectional	None	Interrupts
	8-bit port	2	No handshaking; unusable if port 0 or 1 uses handshaking	Read or write	None	Software polling
PC-DIO-96 PCI-DIO-96 NB-DIO-96	8-bit port	0, 1, 3, 4, 6, 7, 9, 10	Handshaking on or off	Read or write, ports 0, 3, 6, and 9 may be bidirectional	None	Interrupts
	8-bit port	2, 5, 8, 11	No handshaking; unusable if port A and B of the 8255 chip use handshaking	Read or write	None	Software polling
PC-OPDIO-16 (Plug and Play)	Optically-isolated 8-bit port	0, 1	—	Port 0 is output (write); port 1 is input (read)	None	Programmed I/O

Timing Only Devices Hardware Capabilities

Table A-27. Digital Hardware Capabilities—Timing Only Devices

Device	Port Type	Port Numbers	Handshake Modes	Direction	DIO Clocks	Transfer Method
PC-TIO-10 NB-TIO-10	8-bit ports	0, 1	No handshaking	Bit-wise direction control	None	Software polling

Table A-28. Counter/Timer Characteristics—Timing Only Devices

Device	Counter Chip Used	# of General Purpose Counters Available	Timebases Available	# of Bits	Gate Modes Available	Out-puts Avail-able	Output Modes Avail-able	Count Direction
PC-TIO-10 NB-TIO-10	Am-9513	10 (8 have SOURCE inputs at the I/O connector)	Internal: 5 MHz (only on CTR5 and CTR10), 1 MHz, 100 kHz, 10 kHz, 1 kHz, 100 Hz; external	16	high-level, low-level, rising-edge, falling-edge	10	TC pulse, TC toggle	Up or Down

5102 Devices Hardware Capabilities

Table A-29. Analog input configuration programmability

Device	Gain	Coupling
5102 devices	by channel	by channel

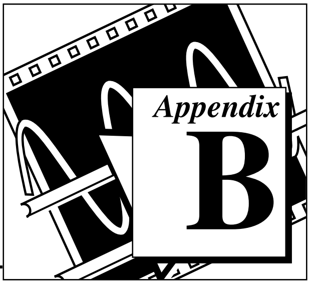
Table A-30. Analog input characteristics

Device	Number of Channels	Resolution	Gains	Range (V)	Input FIFO Words)	Scanning
5102 devices	2	8 bits	1, 5, 20, 100	+/- 5	663,000	1 or 2 channels, in any order without repetitions

Table A-31. Analog input characteristics, Part 2

Device	Triggers	Maximum Sampling Rate (S/s)
5102 devices	SW, Pre, Post, Analog	20,000,000 real time

Multiline Interface Messages



This appendix lists multiline interface messages, which are commands that IEEE 488 defines. Multiline interface message 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. The following list 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*.

Hex	Oct	Dec	ASCII
00	000	0	NUL
01	001	1	SOH
02	002	2	STX
03	003	3	ETX
04	004	4	EOT
05	005	5	ENQ
06	006	6	ACK
07	007	7	BEL
08	010	8	BS
09	011	9	HT
0A	012	10	LF
0B	013	11	VT

Hex	Oct	Dec	ASCII
20	040	32	SP
21	041	33	!
22	042	34	"
23	043	35	#
24	044	36	$
25	045	37	%
26	046	38	&
27	047	39	'
28	050	40	(
29	051	41	)
2A	052	42	*
2B	053	43	+

Hex	Oct	Dec	ASCII
0C	014	12	FF
0D	015	13	CR
0E	016	14	SO
0F	017	15	SI
10	020	16	DLE
11	021	17	DC1
12	022	18	DC2
13	023	19	DC3
14	024	20	DC4
15	025	21	NAK
16	026	22	SYN
17	027	23	ETB
18	030	24	CAN
19	031	25	EM
1A	032	26	SUB
1B	033	27	ESC
1C	034	28	FS
1D	035	29	GS
1E	036	30	RS
1F	037	31	US
40	100	64	@
41	101	65	A

Hex	Oct	Dec	ASCII
2C	054	44	,
2D	055	45	-
2E	056	46	.
2F	057	47	/
30	060	48	0
31	061	49	1
32	062	50	2
33	063	51	3
34	064	52	4
35	065	53	5
36	066	54	6
37	067	55	7
38	070	56	8
39	071	57	9
3A	072	58	:
3B	073	59	;
3C	074	60	<
3D	075	61	=
3E	076	62	>
3F	077	63	?
60	140	96	`
61	141	97	a

Hex	Oct	Dec	ASCII
42	102	66	B
43	103	67	C
44	104	68	D
45	105	69	E
46	106	70	F
47	107	71	G
48	110	72	H
49	111	73	I
4A	112	74	J
4B	113	75	K
4C	114	76	L
4D	115	77	M
4E	116	78	N
4F	117	79	O
50	120	80	P
51	121	81	Q
52	122	82	R
53	123	83	S
54	124	84	T
55	125	85	U
56	126	86	V
57	127	87	W

Hex	Oct	Dec	ASCII
62	142	98	b
63	143	99	c
64	144	100	d
65	145	101	e
66	146	102	f
67	147	103	g
68	150	104	h
69	151	105	i
6A	152	106	j
6B	153	107	k
6C	154	108	l
6D	155	109	m
6E	156	110	n
6F	157	111	o
70	160	112	p
71	161	113	q
72	162	114	r
73	163	115	s
74	164	116	t
75	165	117	u
76	166	118	v
77	167	119	w

Hex	Oct	Dec	ASCII
58	130	88	X
59	131	89	Y
5A	132	90	Z
5B	133	91	[
5C	134	92	\
5D	135	93	]
5E	136	94	^
5F	137	95	_

Hex	Oct	Dec	ASCII
78	170	120	x
79	171	121	y
7A	172	122	z
7B	173	123	{
7C	174	124	|
7D	175	125	}
7E	176	126	~
7F	177	127	DEL

Appendix C

Operation of the GPIB

This appendix 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

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:

- Controller—to manage the GPIB
- Talker—to send data
- Listener—to receive data

The 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.

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).

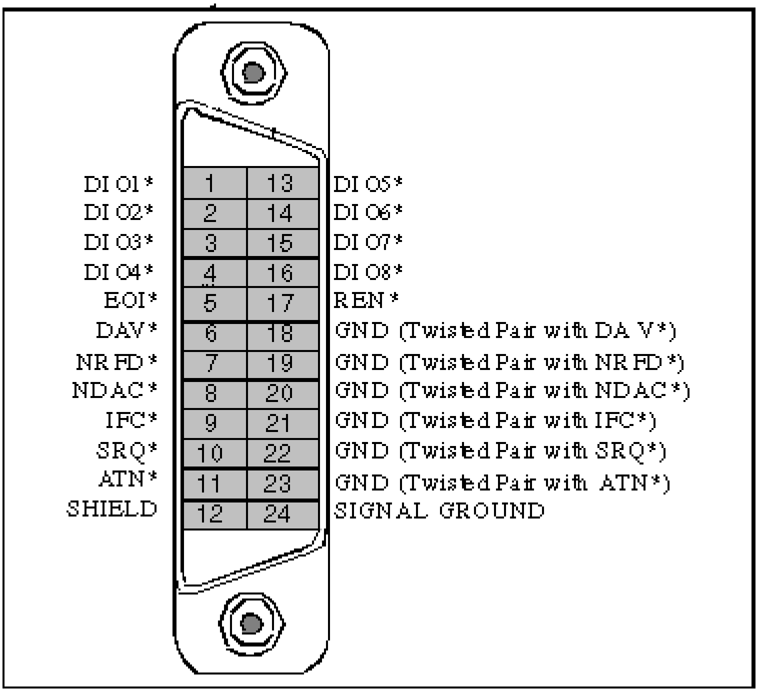

DIO1*
1
13
DIO5*
DIO2*
2
14
DIO6*
DIO3*
3
15
DIO7*
DIO4*
4
16
DIO8*
EOI*
5
17
REN*
DAV*
6
18
GND (Twisted Pair with DAV*)
NRFD*
7
19
GND (Twisted Pair with NRFD*)
NDAC*
8
20
GND (Twisted Pair with NDAC*)
IFC*
9
21
GND (Twisted Pair with IFC*)
SRQ*
10
22
GND (Twisted Pair with SRQ*)
ATN*
11
23
GND (Twisted Pair with ATN*)
SHIELD
12
24
SIGNAL GROUND

Figure C-1. GPIB Connector Showing Signal Assignment

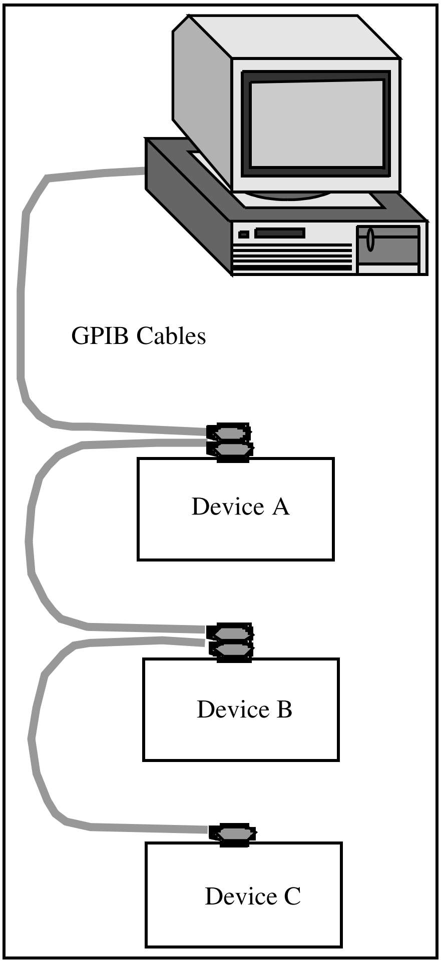


Figure C-2. Linear Configuration

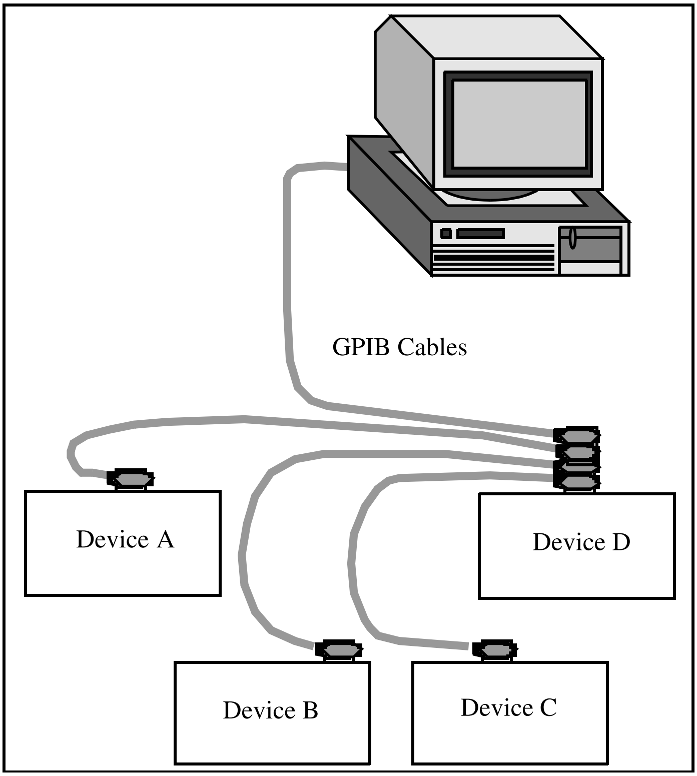


Figure C-3. Star Configuration

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.

Appendix D

References

This appendix lists the reference material used to produce the Analysis VIs in this manual. These references contain more information on the theories and algorithms implemented in the analysis library.

1. Baher, H. *Analog & Digital Signal Processing*. New York: John Wiley & Sons. 1990.
2. Bates, D.M. and Watts, D.G. *Nonlinear Regression Analysis and its Applications*. New York: John Wiley & Sons. 1988.
3. Bracewell, R.N. “Numerical Transforms.” Science. Science-248. 11 May 1990.
4. Burden, R.L. & Faires, J.D. *Numerical Analysis. Third Edition*. Boston: Prindle, Weber & Schmidt. 1985.
5. Chen, C.H. et al. *Signal Processing Handbook*. New York: Marcel Decker, Inc. 1988.
6. DeGroot, M. Probability and Statistics 2nd ed. Reading, Massachusetts: Addison-Wesley Publishing Co. 1986.
7. Dowdy, S. and Wearden, S. *Statistics for Research* 2nd ed. New York: John Wiley & Sons. 1991.
8. Dudewicz, E.J. and Mishra, S.N. *Modern Mathematical Statistics* New York: John Wiley & Sons, 1988.
9. Duhamel, P. et al. “On Computing the Inverse DFT.” IEEE Transactions on ASSP. ASSP-34 (1986): 1 (February).
10. Dunn, O. and Clark, V. *Applied Statistics: Analysis of Variance and Regression* 2nd ed. New York: John Wiley & Sons. 1987.
11. Elliot, D.F. *Handbook of Digital Signal Processing Engineering Applications*. San Diego: Academic Press. 1987.
12. Harris, Fredric J. “On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform,” Proceedings of the IEEE-66 (1978)-1.
13. Maisel, J.E. “Hilbert Transform Works With Fourier Transforms to Dramatically Lower Sampling Rates.” Personal Engineering and Instrumentation News. PEIN-7 (1990): 2 (February).

14. Miller, I. and Freund, J.E. *Probability and Statistics for Engineers*. Englewood Cliffs, N.J.: Prentice-Hall, Inc. 1987.

15. Neter, J. et al. *Applied Linear Regression Models*. Richard D. Irwin, Inc. 1983.

16. Neuvo, Y., Dong, C.-Y., and Mitra, S.K. "Interpolated Finite Impulse Response Filters," IEEE Transactions on ASSP. ASSP-32 (1984): 6 (June).

17. O'Neill, M.A. "Faster Than Fast Fourier." BYTE. (1988) (April).

18. Oppenheim, A.V. & Schafer, R.W. *Discrete-Time Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall. 1989.

19. Parks, T.W. and Burrus, C.S. *Digital Filter Design*. John Wiley & Sons, Inc.: New York. 1987.

20. Pearson, C.E. *Numerical Methods in Engineering and Science*. New York: Van Nostrand Reinhold Co. 1986.

21. Press, W.H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press. 1988.

22. Rabiner, L.R. & Gold, B. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall. 1975.

23. Sorensen, H.V. et al. "On Computing the Split-Radix FFT." IEEE Transactions on ASSP. ASSP-34 (1986):1 (February).

24. Sorensen, H.V. et al. "Real-Valued Fast Fourier Transform Algorithms." IEEE Transactions on ASSP. ASSP-35 (1987): 6 (June).

25. Stoer, J. and Bulirsch, R. *Introduction to Numerical Analysis*. New York: Springer-Verlag. 1987.

26. Vaidyanathan, P.P. *Multirate Systems and Filter Banks*. Englewood Cliffs, New Jersey: Prentice Hall. 1993.

27. Wichman, B. and Hill, D. "Building a Random-Number Generator: A pascal routine for very-long-cycle random-number sequences." BYTE, March 1987, pp. 127-128.

Appendix E

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

United States: (512) 794-5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at (512) 418-1111.

E-Mail Support (currently U.S. only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	Telephone	Fax
Australia	02 9874 4100	02 9874 4455
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 527 2321	09 502 2930
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 5734815	03 5734816
Italy	06 5729961	06 57284309
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	31 348 43 34 66	31 348 43 06 73
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (___)___________________ Phone (___) ______________________________

Computer brand ________________ Model ________________ Processor___________________

Operating system (include version number) ______________________________________

Clock speed ______MHz RAM _____MB _____________________________ Display adapter

Mouse ___yes ___no Other adapters installed_______________________________________

Hard disk capacity _____MB ____Brand

Instruments used ____

____

National Instruments hardware product model __________________________________ Revision

Configuration ____

National Instruments software product ____ Version

Configuration ____

The problem is: ____

____

____

____

____

List any error messages: ____

____

____

The following steps reproduce the problem:____

____

____

____

____

LabVIEW Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

DAQ hardware ____________________

Interrupt level of hardware ____________________

DMA channels of hardware ____________________

Base I/O address of hardware ____________________

Programming choice ____________________

HiQ, NI-DAQ, LabVIEW, or LabWindows version ____________________

Other boards in system ____________________

Base I/O address of other boards ____________________

DMA channels of other boards ____________________

Interrupt level of other boards ____________________

Other Products

Computer make and model ____________________

Microprocessor ____________________

Clock frequency or speed ____________________

Type of video board installed ____________________

Operating system version ____________________

Operating system mode ____________________

Programming language ____________________

Programming language version ____________________

Other boards in system ____________________

Base I/O address of other boards ____________________

DMA channels of other boards ____________________

Interrupt level of other boards ____________________

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **LabVIEW Function and VI Reference Overview**

Edition Date: **May 1997**

Part Number: **321526A-01**

Please comment on the completeness, clarity, and organization of the manual.

____

____

____

____

____

____

____

If you find errors in the manual, please record the page numbers and describe the errors.

____

____

____

____

____

____

____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

Phone (___)____________________________ Fax (___) _______________________________

Mail to:

Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to:

Technical Publications
National Instruments Corporation
(512) 794-5678

Index