

COMPREHENSIVE SERVICES

We offer competitive repair and calibration services, as well as easily accessible documentation and free downloadable resources.

SELL YOUR SURPLUS

We buy new, used, decommissioned, and surplus parts from every NI series. We work out the best solution to suit your individual needs.

 Sell For Cash  Get Credit  Receive a Trade-In Deal

OBSOLETE NI HARDWARE IN STOCK & READY TO SHIP

We stock **New**, **New Surplus**, **Refurbished**, and **Reconditioned** NI Hardware.



Bridging the gap between the manufacturer and your legacy test system.

 1-800-915-6216

 www.apexwaves.com

 sales@apexwaves.com

All trademarks, brands, and brand names are the property of their respective owners.

Request a Quote

 **CLICK HERE**

NB-A0-6



LabVIEW™

Function and VI Reference Manual

Internet Support

E-mail: support@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 (USA)

Tel: 512 795 8248

Fax: 512 794 5678

International Offices

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11,
France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091,
Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, 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,
United Kingdom 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway Austin, Texas 78730-5039 USA 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 THEREOF 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

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

DAQCard™, DAQ-STC™, DAQPad™, LabVIEW™, natinst.com™, National Instruments™, NI-DAQ™, PXI™, RTSI™, and SCXI™, are trademarks of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

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

Organization of the Product User Manual	xxv
Conventions Used in This Manual.....	xxvi
Related Documentation.....	xxvii
Customer Communication	xxviii

Chapter 1

Introduction to the G Functions and VIs

Locating the G Functions and VIs	1-1
Function and VI Overviews	1-2
Structures	1-2
Numeric Functions	1-3
Boolean Functions	1-3
String Functions.....	1-3
Array Functions	1-3
Cluster Functions	1-3
Comparison Functions.....	1-4
Time and Dialog Functions	1-4
File I/O Functions.....	1-4
Advanced Functions	1-4
DAQ	1-5
Instrument I/O	1-5
Communication	1-5
Analysis VIs	1-5
Select A VI.....	1-6
Tutorial	1-6
Instrument Driver Library	1-6
User Library.....	1-6
Application Control.....	1-7

PART I

G Functions and VIs

Chapter 2

G Function and VI Reference Overview

G Functions Overview.....	2-2
Introduction to Polymorphism.....	2-2
Polymorphism	2-2
Unit Polymorphism	2-3
Numeric Conversion	2-4
Overflow and Underflow	2-5
Wire Styles	2-6

Chapter 3

Structures

Structures Overview	3-2
---------------------------	-----

Chapter 4

Numeric Functions

Polymorphism for Numeric Functions	4-2
Polymorphism for Transcendental Functions	4-3
Polymorphism for Conversion Functions	4-3
Polymorphism for Complex Functions	4-4
Arithmetic Function Descriptions	4-4
Conversion Functions Descriptions.....	4-9
Trigonometric and Hyperbolic Functions Descriptions	4-14
Complex Function Descriptions	4-20
Additional Numeric Constants Descriptions	4-21

Chapter 5

Boolean Functions

Polymorphism for Boolean Functions.....	5-1
Boolean Function Descriptions	5-2

Chapter 6

String Functions

Overview of Polymorphism for String Functions.....	6-1
Polymorphism for String Functions	6-1
Polymorphism for Additional String to Number Functions	6-2
Polymorphism for String Conversion Functions	6-2
Format Strings Overview	6-2
String Function Descriptions	6-6
String Conversion Function Descriptions	6-18
String Fixed Constants	6-20

Chapter 7

Array Functions

Array Function Overview	7-2
Out-of-Range Index Values.....	7-3
Polymorphism for Array Functions	7-3
Array Function Descriptions.....	7-3

Chapter 8

Cluster Functions

Cluster Function Overview	8-2
Polymorphism for Cluster Functions	8-3
Setting the Order of Cluster Elements	8-3
Cluster Function Descriptions	8-4

Chapter 9

Comparison Functions

Comparison Function Overview	9-1
Boolean Comparison	9-1
String Comparison.....	9-2
Numeric Comparison.....	9-2
Cluster Comparison	9-2
Comparison Modes.....	9-2
Character Comparison	9-4
Polymorphism for Comparison Functions	9-5
Comparison Function Descriptions	9-6

Chapter 10

Time, Dialog, and Error Functions

Time, Dialog, and Error Functions Overview	10-2
Timing Functions	10-2
Error Handling Overview.....	10-3
Error I/O and the Error State Cluster.....	10-4
Time and Dialog Function Descriptions.....	10-6
Error Handling VI Descriptions	10-10

Chapter 11

File Functions

File I/O VI and Function Overview.....	11-2
High-Level File VIs	11-2
Low-Level File VIs and File Functions	11-2
Byte Stream and Datalog Files.....	11-3
Flow-Through Parameters.....	11-4
Error I/O in File I/O Functions	11-5
Permissions	11-5
File I/O Function and VI Descriptions	11-6
Binary File VI Descriptions.....	11-12
Advanced File Function Descriptions	11-14
Configuration File VIs.....	11-20
File Constants Descriptions.....	11-26

Chapter 12

Application Control Functions

Application Control Functions	12-2
Help Function Descriptions.....	12-7
Menu Functions	12-8

Chapter 13

Advanced Functions

Advanced Function Descriptions	13-2
Data Manipulation Function Descriptions.....	13-4
Memory VI Descriptions	13-7
Synchronization VIs	13-8
Notification VIs	13-8
Queue VIs.....	13-11
Rendezvous VIs.....	13-14

Semaphore VIs.....	13-16
Occurrence Function Descriptions.....	13-19

PART II

Data Acquisition VIs

Chapter 14

Introduction to the LabVIEW Data Acquisition VIs

Finding Help Online for the DAQ VIs	14-2
The Analog Input VIs	14-3
Easy Analog Input VIs	14-4
Intermediate Analog Input VIs.....	14-5
Analog Input Utility VIs.....	14-5
Advanced Analog Input VIs.....	14-5
Locating Analog Input VI Examples.....	14-5
Analog Output VIs.....	14-6
Easy Analog Output VIs.....	14-7
Intermediate Analog Output VIs	14-7
Analog Output Utility VIs	14-7
Advanced Analog Output VIs	14-8
Locating Analog Output VI Examples	14-8
Digital Function VIs	14-8
Easy Digital I/O VIs	14-9
Intermediate Digital I/O VIs.....	14-9
Advanced Digital I/O VIs.....	14-10
Locating Digital I/O VI Examples	14-10
Counter VIs.....	14-10
Easy Counter VIs.....	14-11
Intermediate Counter Input VIs.....	14-11
Advanced Counter VIs	14-12
Locating Counter VI Examples	14-12
Calibration and Configuration VIs	14-12
Signal Conditioning VIs	14-12

Chapter 15

Easy Analog Input VIs

Easy Analog Input VI Descriptions	15-1
-----------------------------------------	------

Chapter 16

Intermediate Analog Input VIs

Handling Errors	16-1
Intermediate Analog Input VI Descriptions	16-2

Chapter 17

Analog Input Utility VIs

Handling Errors	17-2
Analog Input Utility VI Descriptions	17-2

Chapter 18

Advanced Analog Input VIs

Advanced Analog Input VI Descriptions	18-1
---------------------------------------------	------

Chapter 19

Easy Analog Output VIs

Easy Analog Output VI Descriptions	19-1
------------------------------------------	------

Chapter 20

Intermediate Analog Output VIs

Handling Errors	20-1
Analog Output VI Descriptions	20-2

Chapter 21

Analog Output Utility VIs

Handling Errors	21-1
Analog Output Utility VI Descriptions	21-2

Chapter 22

Advanced Analog Output VIs

Advanced Analog Output VI Descriptions	22-1
----------------------------------------------	------

Chapter 23

Easy Digital I/O VIs

Easy Digital I/O Descriptions	23-1
-------------------------------------	------

Chapter 24

Intermediate Digital I/O VIs

Handling Errors.....	24-2
Intermediate Digital I/O VI Descriptions	24-2

Chapter 25

Advanced Digital I/O VIs

Digital Port VI Descriptions	25-2
Digital Group VI Descriptions.....	25-3

Chapter 26

Easy Counter VIs

Easy Counter VI Descriptions	26-2
------------------------------------	------

Chapter 27

Intermediate Counter VIs

Handling Errors.....	27-2
Intermediate Counter VI Descriptions	27-2

Chapter 28

Advanced Counter VIs

Advanced Counter VI Descriptions.....	28-2
---------------------------------------	------

Chapter 29

Calibration and Configuration VIs

Calibration and Configuration VI Descriptions.....	29-2
Channel Configuration VIs.....	29-18

Chapter 30

Signal Conditioning VIs

Signal Conditioning VI Descriptions.....	30-2
------------------------------------------	------

PART III

Instrument I/O Functions and VIs

Chapter 31

Introduction to LabVIEW Instrument I/O VIs

Instrument Drivers Overview	31-2
Instrument Driver Distribution.....	31-3
CD-ROM Instrument Driver Distribution	31-3
Instrument Driver Template VIs	31-4
Introduction to VISA Library	31-4
Introduction to GPIB	31-5
LabVIEW Traditional GPIB Functions	31-5
GPIB 488.2 Functions.....	31-5
Single-Device Functions.....	31-6
Multiple-Device Functions	31-6
Bus Management Functions	31-6
Low-Level Functions.....	31-7
General Functions.....	31-7
Serial Port VI Overview	31-7

Chapter 32

Instrument Driver Template VIs

Introduction to Instrument Driver Template VIs.....	32-1
Instrument Driver Template VI Descriptions.....	32-2

Chapter 33

VISA Library Reference

Operations.....	33-2
VISA Library Reference Parameters	33-2
VISA Operation Descriptions.....	33-4
Event Handling Functions	33-10
High Level Register Access Functions.....	33-12
Low Level Register Access Functions	33-16
VISA Serial Functions.....	33-18
VISA Property Node	33-19
VISA Property Node Descriptions	33-20
Fast Data Channel	33-20
General Settings	33-20
GPIB Settings.....	33-20
Interface Information	33-21

Message-Based Settings	33-21
Modem Line Settings	33-21
PXI Resources	33-21
PXI Settings.....	33-21
Register-Based Settings.....	33-21
Serial Settings.....	33-22
Version Information	33-22
VME/VXE Settings	33-22

Chapter 34

Traditional GPIB Functions

Traditional GPIB Function Parameters.....	34-2
Traditional GPIB Function Behavior.....	34-3
Traditional GPIB Function Descriptions	34-3
GPIB Device and Controller Functions	34-7
Device Functions	34-7
Controller Functions	34-9

Chapter 35

GPIB 488.2 Functions

GPIB 488.2 Common Function Parameters	35-1
GPIB 488.2 Function Descriptions (Single-Device Functions).....	35-2
GPIB 488.2 Multiple-Device Function Descriptions	35-4
GPIB 488.2 Bus Management Function Descriptions.....	35-6
GPIB 488.2 Low-Level I/O Function Descriptions.....	35-8
GPIB 488.2 General Function Descriptions	35-10

Chapter 36

Serial Port VIs

Serial Port VI Descriptions	36-1
-----------------------------------	------

PART IV Analysis VIs

Chapter 37

Introduction to Analysis in LabVIEW

Full Development System.....	37-2
Analysis VI Overview	37-2

Analysis VI Organization 37-3
Notation and Naming Conventions 37-4

Chapter 38

Signal Generation VIs

Signal Generation VI Descriptions 38-2

Chapter 39

Digital Signal Processing VIs

Signal Processing VI Descriptions 39-2

Chapter 40

Measurement VIs

Measurement VI Descriptions 40-2

Chapter 41

Filter VIs

Filter VI Descriptions 41-2

Chapter 42

Window VIs

Window VI Descriptions 42-2

Chapter 43

Curve Fitting VIs

Curve Fitting VI Descriptions 43-2

Chapter 44

Probability and Statistics VIs

Probability and Statistics VI Descriptions 44-2

Chapter 45

Linear Algebra VIs

Linear Algebra VI Descriptions 45-2

Chapter 46

Array Operation VIs

Array Operation VI Descriptions.....	46-2
--------------------------------------	------

Chapter 47

Additional Numerical Method VIs

Additional Numerical Method VI Descriptions.....	47-1
--------------------------------------------------	------

PART V

Communication VIs and Functions

Chapter 48

TCP VIs

TCP VI Description	48-2
TCP/IP Functions.....	48-2

Chapter 49

UDP VIs

UDP VI Descriptions	49-1
---------------------------	------

Chapter 50

DDE VIs

DDE Client VI Descriptions	50-2
DDE Server VI Descriptions	50-3

Chapter 51

ActiveX Automation Functions

ActiveX Automation Function Descriptions	51-2
Data Conversion Function	51-4

Chapter 52

AppleEvent VIs

General AppleEvent VI Behavior.....	52-2
The User Identity Dialog Box	52-2
Target ID	52-3
Send Options	52-4
Targeting VI Descriptions	52-4

AppleEvent VI Descriptions.....	52-6
LabVIEW-Specific AppleEvent VIs	52-8
Advanced Topics	52-9
Constructing and Sending Other AppleEvents	52-9
Creating AppleEvent Parameters	52-10
Low-Level AppleEvent VIs	52-13
Object Support VI Example	52-16
Sending AppleEvents to LabVIEW from Other Applications	52-18
Required AppleEvents	52-18
LabVIEW Specific AppleEvents	52-18
Replies to AppleEvents.....	52-18
Event: Run VI.....	52-19
Description	52-19
Event Class.....	52-19
Event ID	52-19
Event Parameters.....	52-19
Reply Parameters.....	52-19
Possible Errors.....	52-19
Event: Abort VI	52-20
Description	52-20
Event Class.....	52-20
Event ID	52-20
Event Parameters.....	52-20
Reply Parameters.....	52-20
Possible Errors.....	52-20
Event: VI Active?	52-21
Description	52-21
Event Class.....	52-21
Event ID	52-21
Event Parameters.....	52-21
Reply Parameters.....	52-21
Possible Errors.....	52-21
Event: Close VI	52-22
Description	52-22
Event Class.....	52-22
Event ID	52-22
Event Parameters.....	52-22
Reply Parameters.....	52-22
Possible Errors.....	52-22

Chapter 53 Program to Program Communication VIs

PPC VI Descriptions	53-2
---------------------------	------

Appendices and Index

Appendix A Error Codes

Numeric Error Codes	A-1
---------------------------	-----

Appendix B DAQ Hardware Capabilities

MIO and AI Device Hardware Capabilities.....	B-1
Lab and 1200 Series and Portable Devices Hardware Capabilities.....	B-10
54xx Devices.....	B-14
SCXI Module Hardware Capabilities	B-16
Analog Output Only Devices Hardware Capabilities	B-20
Dynamic Signal Acquisition Devices Hardware Capabilities	B-21
Digital Only Devices Hardware Capabilities.....	B-22
Timing Only Devices Hardware Capabilities	B-23
5102 Devices Hardware Capabilities.....	B-24

Appendix C GPIB Multiline Interface Messages

Multiline Interface Messages.....	C-1
Message Definitions	C-6

Appendix D Customer Communication

Index

Figures and Tables

Figures

Figure 27-1.	Setup Mode in ICTR Control.....	27-5
Figure 27-2.	Setup Mode 1 in ICTR Control.....	27-6
Figure 27-3.	Setup Mode 2 in ICTR Control.....	27-6
Figure 27-4.	Setup Mode 3 in ICTR Control.....	27-6
Figure 27-5.	Setup Mode 4 in ICTR Control.....	27-6
Figure 27-6.	Setup Mode 5 in ICTR Control.....	27-7
Figure 28-1.	Unbuffered Mode 2 and 3 Counting	28-4
Figure 28-2.	Buffered Mode 3 Counting	28-5
Figure 28-3.	Unbuffered Mode 4 High Pulse Width Measurement	28-6
Figure 28-4.	Buffered Mode 4 Rising-Edge Pulse Width Measurement.....	28-6
Figure 28-5.	Unbuffered Mode 4 Rising-Edge Period Measurement.....	28-7
Figure 28-6.	Buffered Mode 4 Rising-Edge Pulse Width Measurement.....	28-7
Figure 28-7.	Unbuffered Mode 6 High Pulse Width Measurement	28-7
Figure 28-8.	Buffered Mode 6 High Pulse Width Measurement (Count on Rising Edge of Source)	28-8
Figure 28-9.	Buffered Mode 7 Semi-Period Measurement	28-8
Figure 30-1.	Strain Gauge Bridge Completion Networks (Quarter-Bridge Configuration)	30-4
Figure 30-2.	Strain Gauge Bridge Completion Networks (Half-Bridge Configuration)	30-5
Figure 30-3.	Strain Gauge Bridge Completion Networks (Full-Bridge Configuration).....	30-6
Figure 30-4.	Circuit Diagram of a Thermistor in a Voltage Divider.....	30-7
Figure 30-5.	Circuit Diagram of a Thermistor with Current Excitation.....	30-7
Figure 41-1.	Lowpass Filter.....	41-8
Figure 41-2.	Highpass Filter.....	41-8
Figure 41-3.	Bandpass Filter.....	41-8
Figure 41-4.	Bandstop Filter.....	41-9

Tables

Table 6-1.	Special Escape Codes	6-3
Table 6-2.	String Syntax.....	6-4
Table 6-3.	Possible Format into String Errors.....	6-7
Table 6-4.	Format Specifiers	6-7
Table 6-5.	Special Characters for Match Pattern	6-9
Table 6-6.	Strings for the Match Pattern Examples	6-10

Table 6-7.	Scan from String Errors.....	6-12
Table 6-8.	Scan from String Examples.....	6-12
Table 9-1.	Lexical Class Number Descriptions	9-8
Table 10-1.	Valid Value of Elements for Date/Time Cluster	10-2
Table 10-2.	Format Codes for the Time Format String	10-7
Table 18-1.	AI Buffer Config VI Device-Specific Settings and Ranges.....	18-2
Table 18-2.	Device-Specific Settings and Ranges for Controls in the AI Clock Config VI.....	18-4
Table 18-3.	Device-Specific Settings and Ranges for the AI Control VI.....	18-6
Table 18-4.	Device-Specific Settings and Ranges for the AI Group Config VI	18-7
Table 18-5.	AI Hardware Config Channel Configuration	18-9
Table 18-6.	Device-Specific Settings and Ranges for the AI Hardware Config VI...	18-11
Table 18-7.	Device-Specific Settings and Ranges for the AI SingleScan VI.....	18-14
Table 18-8.	Restrictions for Analog Triggering on E-Series Devices	18-17
Table 18-9.	Digital Trigger Sources for Devices with Fixed Digital Trigger Sources	18-18
Table 18-10.	Device-Specific Settings and Ranges for the AI Trigger Config VI (Part 1).....	18-18
Table 18-11.	Device-Specific Settings and Ranges for the AI Trigger Config VI (Part 2).....	18-20
Table 18-12.	Device-Specific Settings and Ranges for the AI Trigger Config VI (Part 3).....	18-20
Table 18-13.	Device-Specific Settings and Ranges for the AI Trigger Config VI (Part 4).....	18-21
Table 25-1.	Device Specific Parameters and Legal Ranges for Devices.....	25-6
Table 28-1.	Counter Chips and Their Available DAQ Devices	28-2
Table 28-2.	Valid Counter Numbers for CTR Group Config Devices	28-3
Table 28-3.	Adjacent Counters	28-9
Table 29-1.	Channel to Index VI Parameter Examples	29-8
Table 29-2.	Channel to Index VI Parameter Examples for Sun	29-9
Table 34-1.	Command String Device Functions	34-4
Table 34-2.	Command String Controller Functions	34-4
Table 51-1.	New and Old ActiveX Automation Functions	51-2
Table 52-1.	AppleEvent Descriptor String Formats	52-11

Table A-1.	Numeric Error Code Ranges	A-1
Table A-2.	VISA Error Codes	A-2
Table A-3.	Analysis Error Codes	A-4
Table A-4.	Data Acquisition VI Error Codes	A-7
Table A-5.	AppleEvent Error Codes	A-21
Table A-6.	Instrument Driver Error Codes	A-22
Table A-7.	PPC Error Codes	A-23
Table A-8.	GPIB Error Codes	A-24
Table A-9.	LabVIEW Function Error Codes	A-25
Table A-10.	LabVIEW-Specific PPC Error Codes	A-28
Table A-11.	TCP and UDP Error Codes	A-28
Table A-12.	Serial Port Error Codes	A-29
Table A-13.	LabVIEW-Specific Error Codes for AppleEvent Messages.....	A-29
Table A-14.	DDE Error Codes	A-29
Table B-1.	Analog Input Configuration Programmability—MIO and AI Devices ..	B-1
Table B-2.	Analog Input Characteristics—MIO and AI Devices (Part 1).....	B-2
Table B-3.	Analog Input Characteristics—MIO and AI Devices (Part 2).....	B-3
Table B-4.	Internal Channel Support—MIO and AI Devices	B-4
Table B-5.	Analog Output Characteristics—MIO and AI Devices	B-4
Table B-6.	Analog Output Characteristics—E Series Devices.....	B-7
Table B-7.	Digital I/O Hardware Capabilities—MIO and AI Devices.....	B-8
Table B-8.	Counter Characteristics—MIO and AI Devices	B-9
Table B-9.	Counter Usage for Analog Input and Output—MIO and AI Devices	B-10
Table B-10.	Analog Input Configuration Programmability— Lab and 1200 Series and Portable Devices	B-10
Table B-11.	Analog Input Characteristics—Lab and 1200 Series and Portable Devices (Part 1)	B-11
Table B-12.	Analog Input Characteristics—Lab and 1200 Series and Portable Devices (Part 2)	B-11
Table B-13.	Analog Output Characteristics—Lab and 1200 Series and Portable Devices	B-12
Table B-14.	Counter Usage for Analog Input and Output—Lab Series and Portable Devices	B-12
Table B-15.	Digital I/O Hardware Capabilities—Lab and 1200 Series and Portable Devices	B-13
Table B-16.	Analog Output and Digital Output Characteristics— 54XX Series Devices	B-14
Table B-17.	Counter/Timer Characteristics—Lab and 1200 Series and Portable Devices	B-15
Table B-18.	Analog Input Characteristics—SCXI Modules (Part 1)	B-16
Table B-19.	Analog Output Characteristics—SCXI Modules	B-17
Table B-20.	Relay Characteristics—SCXI Modules	B-17

Table B-21.	Digital Input and Output Characteristics—SCXI Modules.....	B-18
Table B-22.	Terminal Block Selection Guide—SCXI Modules	B-18
Table B-23.	Analog Input Configuration Programmability	B-19
Table B-24.	Analog Input Configuration Programmability	B-19
Table B-25.	Analog Output Characteristics—Analog Output Only Devices.....	B-20
Table B-26.	Analog Input Configuration Programmability— Dynamic Signal Acquisition Devices.....	B-21
Table B-27.	Analog Output Characteristics— Dynamic Signal Acquisition Devices.....	B-21
Table B-28.	Analog Input Characteristics— Dynamic Signal Acquisition Devices.....	B-22
Table B-29.	Digital Hardware Capabilities—Digital I/O Devices.....	B-22
Table B-30.	Digital Hardware Capabilities—Timing Only Devices	B-23
Table B-31.	Counter/Timer Characteristics—Timing Only Devices.....	B-24
Table B-32.	Analog Input Configuration Programmability	B-24
Table B-33.	Analog Input Characteristics	B-24
Table B-34.	Analog Input Characteristics, Part 2.....	B-24

About This Manual

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

- 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 you should be familiar with that material.

This manual provides an overview of each function and VI available in the LabVIEW development system. However, for more specific parameter information regarding each function and VI, refer to the *Online Reference*, which you can access by selecting **Help»Online Reference**, or to the Help window, which you access by selecting **Help»Show Help**.

Organization of the Product User Manual

This manual covers five subject areas: G functions and VIs, Data Acquisition VIs, Instrument I/O VIs, Analysis VIs, and Communications VIs. Chapter 1, *Introduction to the G Functions and VIs*, introduces the functions and VIs available in the LabVIEW development system.

- Part I, *G Functions and VIs*, includes Chapters 2 through 13, which describe the functions unique to the G programming language.
- Part II, *Data Acquisition VIs*, includes Chapters 14 through 30, which describe the Data Acquisition (DAQ) VIs.
- Part III, *Instrument I/O Functions and VIs*, includes Chapters 31 through 36, which describe the Instrument I/O VIs and functions.
- Part IV, *Analysis VIs*, includes Chapters 37 through 47, which describe the Analysis VIs.
- Part V, *Communication VIs and Functions*, includes Chapters 48 through 53, which describe the Communication VIs.

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

- Appendix A, *Error Codes*, includes tables that summarize the analog and digital I/O capabilities of National Instruments data acquisition devices.
- Appendix B, *DAQ Hardware Capabilities*, lists commands that IEEE 488 defines.
- Appendix C, *GPIB Multiline Interface Messages*, describes basic concepts you need to understand to operate GPIB.
- Appendix D, *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, <shift>. 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. |
| bold | Bold text denotes the names of menus, menu items, parameters, dialog boxes, dialog box buttons or options, icons, windows, Windows 95 tabs, or LEDs. |
| <i>bold italic</i> | Bold italic text denotes an activity objective, note, caution, or warning. |
| Ctrl | Key names are capitalized. |

<i>italic</i>	Italic text denotes variables, 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.
<i>italic monospace</i>	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 you 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.
monospace bold	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.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.

Related Documentation

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

- *LabVIEW User Manual*
- *G Programming Reference Manual*
- *LabVIEW Data Acquisition Basics Manual*
- *LabVIEW QuickStart Guide*
- *LabVIEW Online Reference*, available by selecting **Help»Online Reference**
- *LabVIEW Online Tutorial (Windows only)*, which you launch from the LabVIEW dialog box.
- *LabVIEW Getting Started Card*
- *G Programming Quick Reference Card*
- *LabVIEW Release Notes*
- *LabVIEW Upgrade Notes*

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 D, *Customer Communication*, at the end of this manual.

Introduction to the G Functions and VIs

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

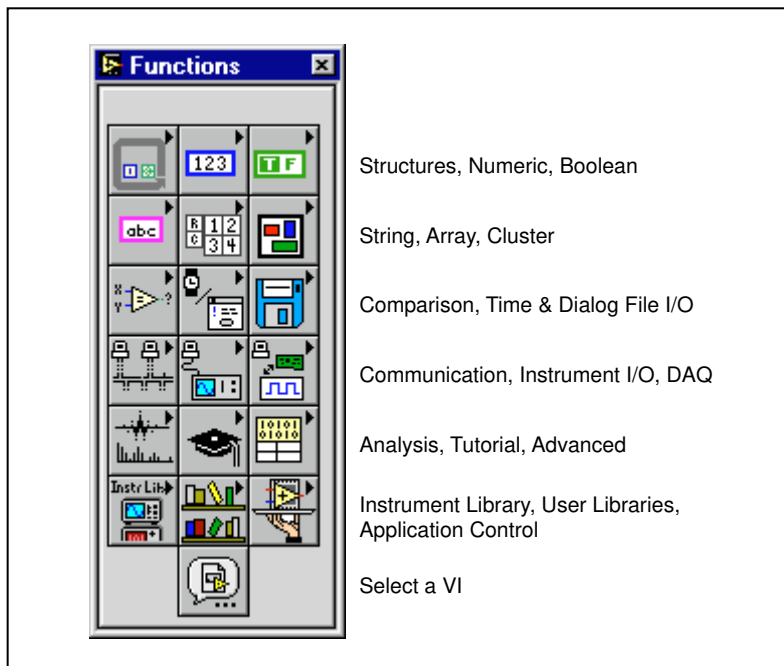
The development system includes collections of VIs that work with your G programming language, data acquisition (DAQ) hardware devices, instrument devices, and other communication interfaces.

Locating the G Functions and VIs

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.

The following illustration shows the functions and VIs available from the Functions palette.



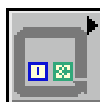
Many **Functions** palette chapters include information about function examples. The paths for these examples for LabVIEW begin with `examples\`.

Function and VI Overviews

The following functions and VIs are available from the **Functions** palette.

Structures

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



Numeric Functions

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



Boolean Functions

Boolean functions perform Boolean and logical operations.



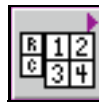
String Functions

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



Array Functions

Array functions assemble, disassemble, and process arrays.



Cluster Functions

Cluster functions 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 a group or array of values.



Time and Dialog Functions

Time and Dialog functions manipulate time functions and display dialog boxes. This palette also includes the VIs 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 are highly specialized. The Code Interface Node is an example of an advanced function. The **Advanced** palette also contains 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 14, [Introduction to the LabVIEW Data Acquisition VIs](#), for more information.



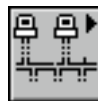
Instrument I/O

Instrument I/O VIs communicate with instruments using GPIB, VISA, or serial communication. See Chapter 31, [Introduction to LabVIEW Instrument I/O VIs](#), for more information.



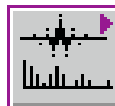
Communication

Communication VIs network to other applications using TCP/IP, DDE, ActiveX, Apple Events, PPC, or UDP. See Chapter 48, [TCP VIs](#), through Chapter 53, [Program to Program Communication VIs](#), 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 37, [Introduction to Analysis in LabVIEW](#), for more information.



Select A VI...

The **Select a VI...** allows you to select any VI using a file dialog box and then place it on a diagram.



Tutorial

The Tutorial VIs provide examples for you to use while working through the *LabVIEW User 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.



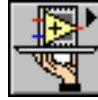
User Library

The **User Library** 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.



Application Control

The **Application Control** palette includes the Help functions, Menu functions, Print VIs, and VI Server VIs.



G Functions and VIs

Part I, *G Functions and VIs*, introduces the G Functions and VIs descriptions. This part contains the following chapters:

- Chapter 2, *G Function and VI Reference Overview*, introduces the G functions and VIs. This chapter also describes the differences between functions and VIs.
- Chapter 3, *Structures*, describes the structures available in G.
- Chapter 4, *Numeric Functions*, 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 as additional numeric constants.
- Chapter 5, *Boolean Functions*, describes the functions that perform logical operations.
- Chapter 6, *String Functions*, describes the string functions, including those that convert strings to numbers and numbers to strings.
- Chapter 7, *Array Functions*, describes the functions for array operations.
- Chapter 8, *Cluster Functions*, describes the functions for cluster operations.
- Chapter 9, *Comparison Functions*, describes the functions that perform comparisons or conditional tests.
- Chapter 10, *Time, Dialog, and Error Functions*, 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.
- Chapter 11, *File Functions*, describes the low-level VIs and functions that manipulate files, directories, and paths. This chapter also describes file constants and the high-level file VIs.

- Chapter 12, *Application Control Functions*, describes the Application Control functions.
- Chapter 13, *Advanced Functions*, describes the functions that perform advanced operations. This chapter also describes the Help, Data Manipulation, and Synchronization functions, and the VI Control and Memory VISA.

G Function and VI Reference Overview

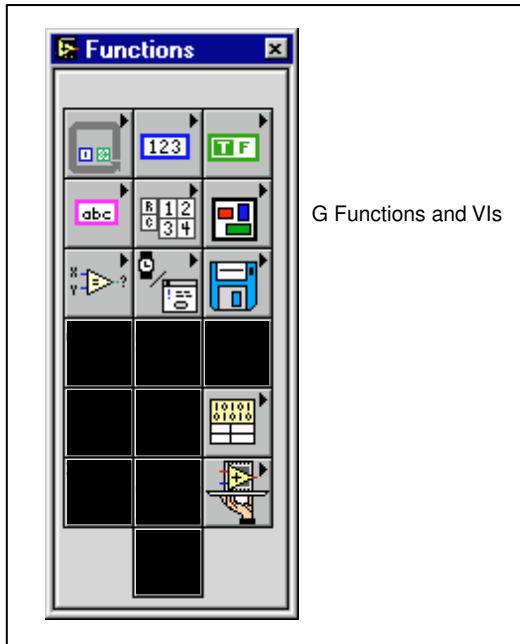
This chapter introduces the G Functions and VIs, descriptions of which comprise Chapter 3 through Chapter 13.

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 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.

The following illustration shows the G functions and VIs available on the **Functions** palette.



Many **Functions** palette chapters include information about function examples.

G Functions Overview

For brief descriptions of each of the eleven G Function and VI palettes available, refer to Chapter 1, *Introduction to the G 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 $\$1 \text{ s}$ 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 $\$1 \ \$2 / \text{s}$. 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 $\text{kg m} / \text{s}^2$.

Suppose a different VI has two inputs of $\$1$ and $\$/s$, 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 $\$/s$ 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 $\$/s$ 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 or no execution time, and overflow much more easily. The analysis libraries, for example, use double-precision, floating-point numbers. You should only use extended-precision, floating-point numbers when necessary. The performance and precision of extended-precision arithmetic varies among the platforms.

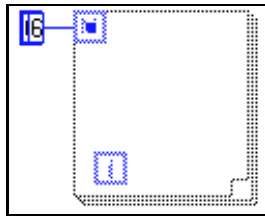
For integers, it is usually best to use a long integer.

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

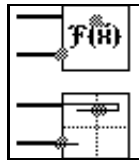
- Signed or unsigned integer to floating-point number—Conversion is exact, except for long integers to single-precision, floating-point numbers. In this case, G reduces the precision from 32 bits to 24 bits.
- Floating-point number to signed or unsigned integer—G moves out-of-range values to the integer's minimum or maximum value. Most integer objects, such as the iteration terminal of a For Loop, round floating-point numbers. G rounds a fractional part of 0.5 to the nearest even integer—for example, G rounds 6.5 to 6 rather than 7.

- Integer to integer—G does not move out-of-range values to the integer's minimum or maximum value. If the source is smaller than the destination, G 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, G copies only the low order bits of the value.

The block diagram 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 increase its execution time. You should try to keep data types consistent in your VIs.

Overflow and Underflow

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

Floating-point operations propagate not-a-number (NaN) and $\pm\text{Inf}$ faithfully. When you explicitly or implicitly convert NaN or $\pm\text{Inf}$ to an integer or Boolean value, however, you get a value that looks reasonable, but is meaningless. For example, dividing by zero produces $\pm\text{Inf}$, 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

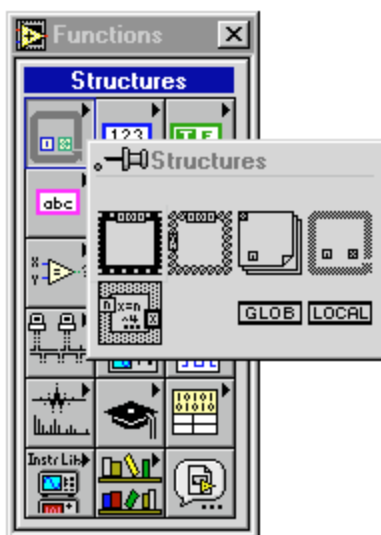
The wire style represents the data type for each terminal, as the following table shows. Polymorphic functions show the wire style for the most commonly used data type.

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

Structures

This chapter describes the Structures available in G.

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



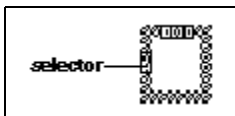
See `examples\general\structs.llb` for examples of how these structures are used in LabVIEW.

Structures Overview

The following Structures are available in G.

Case Structure

Has one or more subdiagrams, or *cases*, exactly one of which executes when the structure executes. Whether it executes depends on the value of the Boolean, string, 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 4, *Case and Sequence Structures and the Formula Node*, in the *LabVIEW User Manual*.

Sequence Structure

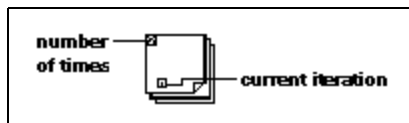
Consists of one or more subdiagrams, or *frames*, that execute sequentially. As an option, you can add sequence locals that allow you to pass information from one frame to subsequent frames by popping up on the edge of the structure.



For more information on how to use the Sequence structure in LabVIEW, see Chapter 4, *Case and Sequence Structures and the Formula Node*, in the *LabVIEW User Manual*.

For Loop

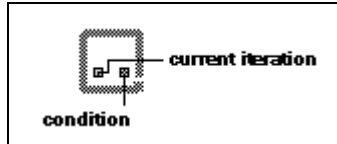
Executes its subdiagram n times, where n equals the value contained in the count terminal. As an option, you can add shift registers so that you can pass information from one iteration to the next by popping up on the edge of the structure.



For more information on how to use For Loop in LabVIEW, see Chapter 3, *Loops and Charts*, in the *LabVIEW User Manual*.

While Loop

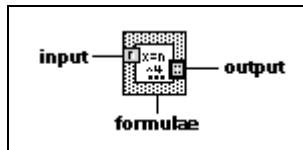
Executes its subdiagram until a Boolean value you wire to the *conditional terminal* is FALSE. As an option, you can add shift registers so you can pass information from one iteration to the next by popping up on the edge of the structure.



For more information on how to use While Loop in LabVIEW, see Chapter 3, *Loops and Charts*, in the *LabVIEW User Manual*.

Formula Node

Executes mathematical formulae on the block diagram.



For more information on the Formula Node, see Chapter 4, *Case and Sequence Structures and 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. You can read and write values to the global variable.



For more information on the Global Variable, see Chapter 23, *Global and Local Variables*, in the *G Programming Reference Manual*.

Local Variable

Lets you read or write to 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.

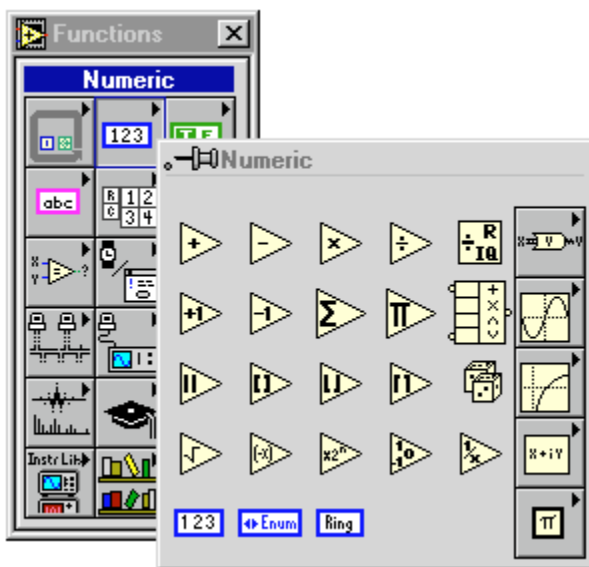
LOCAL

For more information on the Local Variable, see Chapter 23, *Global and Local Variables*, in the *G Programming Reference Manual*.

Numeric Functions

This chapter describes the functions that perform arithmetic, complex, conversion, logarithmic, and trigonometric operations. It also describes the commonly used constants such as the Numeric constant, Enumerated constant, and Ring constant, as well as 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.



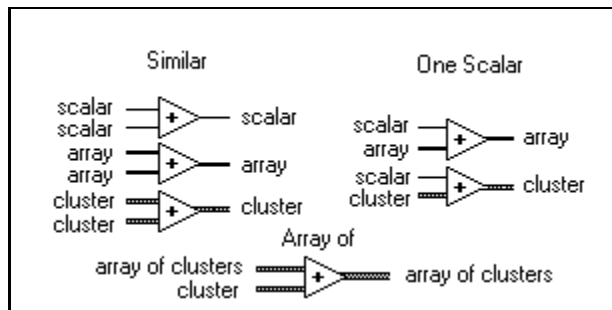
Note

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.

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



Polymorphism for Transcendental Functions

The transcendental 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. Therefore, 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.

When you compare signed and unsigned integers and the signed integer is negative, the negative integer is changed to positive before the comparison occurs. Therefore, you do not get the expected results. For example, if you enter -1 with representation I32 for one input and 5 with a representation U32 as the other input, the result returned states that the minimum value is 5 , because 5 is less than 4294967295 .

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.

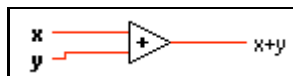
Absolute Value

Returns the absolute value of the input.



Add

Computes the sum of the inputs.



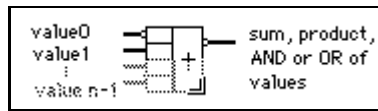
Add Array Elements

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**.

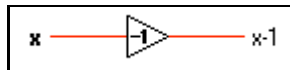
You can invert the inputs or the output of this function by popping up on the individual terminals, and selecting **Invert**. For Add, select **Invert** to negate an input or the output. For Multiply, select **Invert** to use the reciprocal of an input or to produce the reciprocal of the output. For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

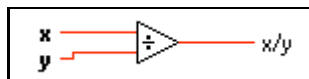
Decrement

Subtracts 1 from the input value.



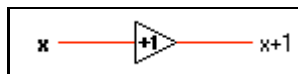
Divide

Computes the quotient of the inputs.



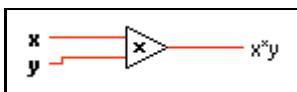
Increment

Adds 1 to the input value.



Multiply

Returns the product of the inputs.



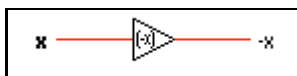
Multiply Array Elements

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



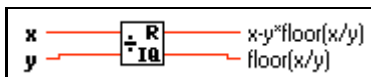
Negate

Negates the input value.



Quotient & Remainder

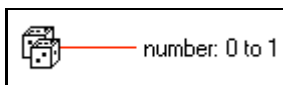
Computes the integer quotient and the remainder of the inputs.



If the integer input value of y is zero, the quotient is zero and the remainder is 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.



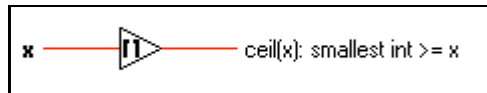
Reciprocal

Divides 1 by the input value.



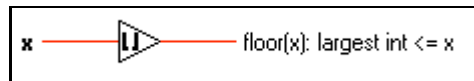
Round To +Infinity

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 .



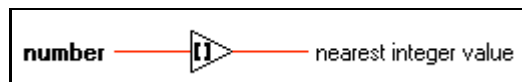
Round To -Infinity

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 .



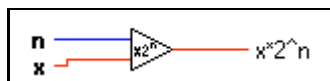
Round To Nearest

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).



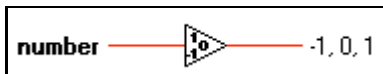
Scale By Power Of 2

Multiplies one input (x) by 2 raised to the power of the other input (n). If n is a floating-point number, 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.



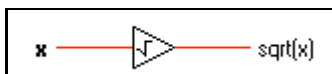
Sign

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.



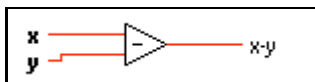
Square Root

Computes the square root of the input value. If x is negative, the square root is NaN unless x is complex.



Subtract

Computes the difference of the inputs.



User Definable Arithmetic Constants

You can define the following constants.

Numeric Constant

123

Use this constant to supply a constant numeric value to the block diagram. Set this value by clicking in the constant with the Operating tool and typing 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

Enum

Enumerated values associate unsigned integers to strings. If you display a value from an enumerated constant, the string is displayed, instead of the number associated with it. If you need a set of strings that do not change, then use this constant. Set the value by clicking in the constant with the Operating Tool. Set the string with the Labeling Tool and enter the string. To add another item, click 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

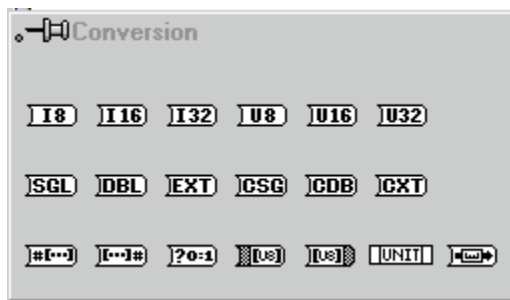


Rings associate unsigned integers to strings. If you display a value from a ring constant, the number is displayed, instead of the string associated with it. If you need a set of strings that do not change, then use this constant. Set the value by clicking the constant with the Operating tool. Set the 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:

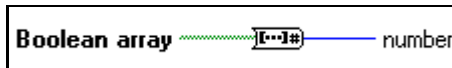
- To Byte Integer
- To Double Precision Complex
- To Double Precision Float
- To Extended Complex
- To Extended Precision Float
- To Long Integer
- To Single Precision Complex
- To Single Precision Float
- To Unsigned Byte Integer
- To Unsigned Word Integer
- To Unsigned Long Integer
- To Word Integer

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.

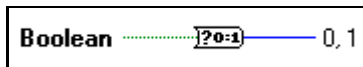
Boolean Array To Number

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)

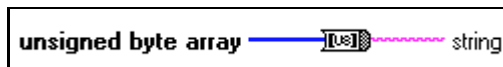
Converts a Boolean value to a word integer—0 and 1 for the input values FALSE and TRUE, respectively.



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](#).

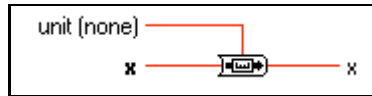
Byte Array To String

Converts an array of unsigned bytes into a string.



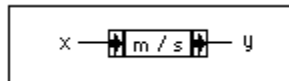
Cast Unit Bases

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 caution. Because the Cast Unit Bases function works with bases, you must understand the conversion from an arbitrary unit to its bases before you can use this function effectively. 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 the unit by highlighting the string with the Operating tool 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 **unit** is meters, the result is 37 with no associated units. If **unit** is feet (ft), the result is 121.36 with no associated units.

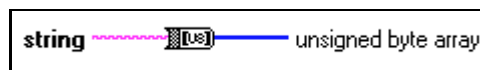
Number To Boolean Array

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.



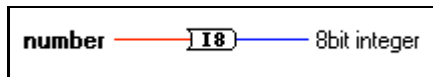
String To Byte Array

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



To Byte Integer

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



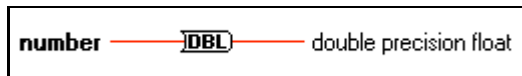
To Double Precision Complex

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



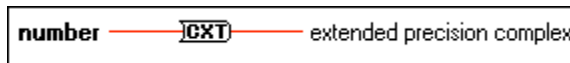
To Double Precision Float

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



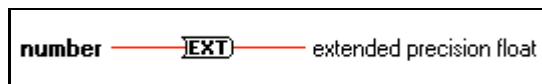
To Extend Precision Complex

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



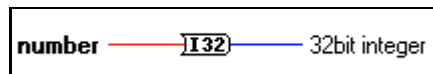
To Extended Precision Float

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 $2^{31}-1$.



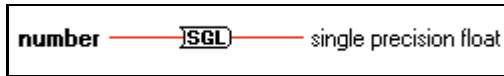
To Single Precision Complex

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



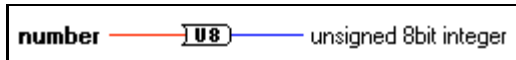
To Single Precision Float

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.



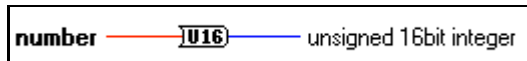
To Unsigned Long Integer

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



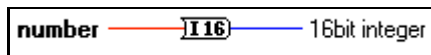
To Unsigned Word Integer

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



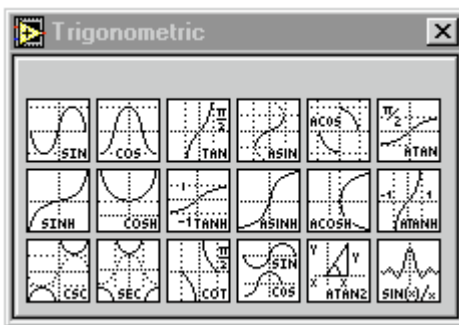
To Word Integer

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



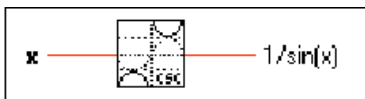
Trigonometric and Hyperbolic Functions Descriptions

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



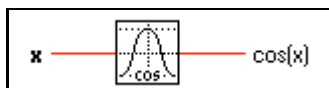
Cosecant

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



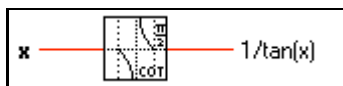
Cosine

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



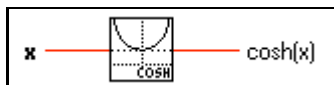
Cotangent

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



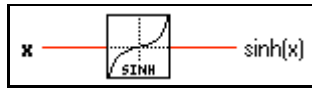
Hyperbolic Cosine

Computes the hyperbolic cosine of x .



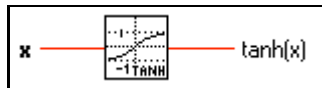
Hyperbolic Sine

Computes the hyperbolic sine of x .



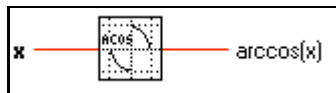
Hyperbolic Tangent

Computes the hyperbolic tangent of x .



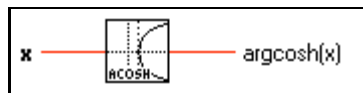
Inverse Cosine

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.



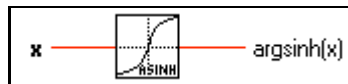
Inverse Hyperbolic Cosine

Computes the hyperbolic arccosine of x . If x is not complex and is less than 1 , the result is NaN.



Inverse Hyperbolic Sine

Computes the hyperbolic argsine of x .



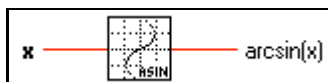
Inverse Hyperbolic Tangent

Computes the hyperbolic arctangent of x . If x is not complex and is less than -1 or greater than 1 , the result is NaN.



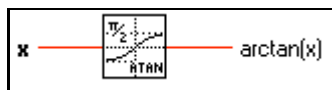
Inverse Sine

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.



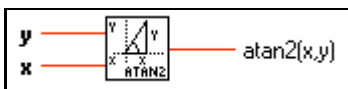
Inverse Tangent

Computes the arctangent of x in radians (which can be between $-\pi/2$ and $\pi/2$).



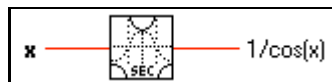
Inverse Tangent (2 Input)

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.



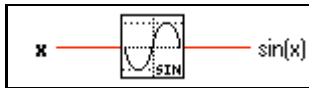
Sinc

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



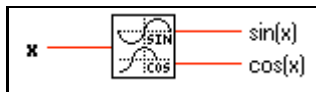
Sine

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



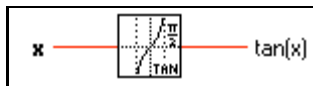
Sine & Cosine

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



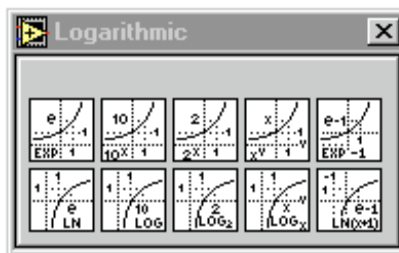
Tangent

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



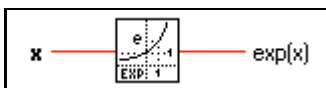
Logarithmic Functions Descriptions

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



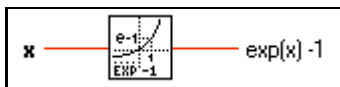
Exponential

Computes the value of e raised to the x power.



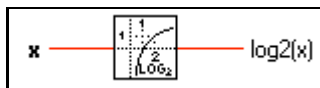
Exponential (Arg) -1

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 then subtracting 1 from the output.



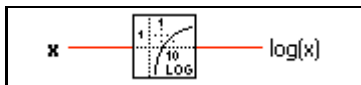
Logarithm Base 2

Computes the base-2 logarithm of x . If x is 0, $\log_2(x)$ is $-\infty$. If x is not complex and is less than 0, $\log_2(x)$ is NaN.



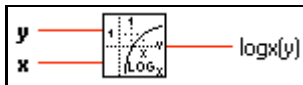
Logarithm Base 10

Computes the base-10 logarithm of x . If x is 0, $\log(x)$ is $-\infty$. If x is not complex and is less than 0, $\log(x)$ is NaN.



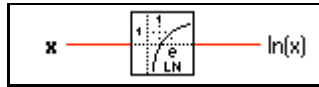
Logarithm Base X

Computes the base x logarithm of y ($x > 0$, $y > 0$). If y is 0, the output is $-\infty$. 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.



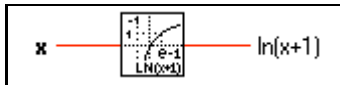
Natural Logarithm

Computes the natural base e logarithm of x . If x is 0, $\ln(x)$ is $-\infty$. If x is not complex and is less than 0, $\ln(x)$ is NaN.



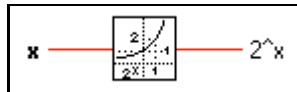
Natural Logarithm (Arg +1)

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



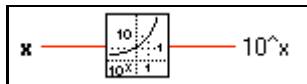
Power Of 2

Computes 2 raised to the x power.



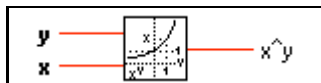
Power Of 10

Computes 10 raised to the x power.



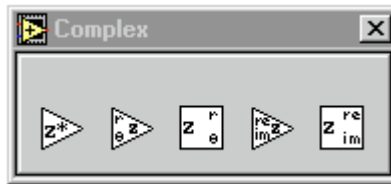
Power Of X

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

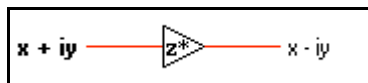
The following illustration displays the options available on the **Complex** subpalette.



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

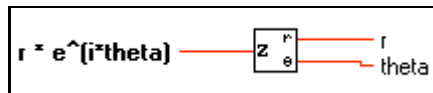
Complex Conjugate

Produces the complex conjugate of $x + iy$.



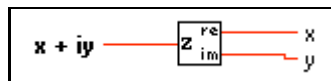
Complex To Polar

Breaks a complex number into its polar components.



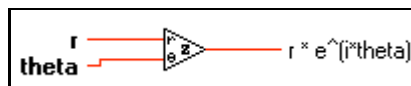
Complex To Re/Im

Breaks a complex number into its rectangular components.



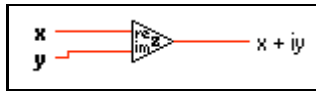
Polar To Complex

Creates a complex number from two values in polar notation.



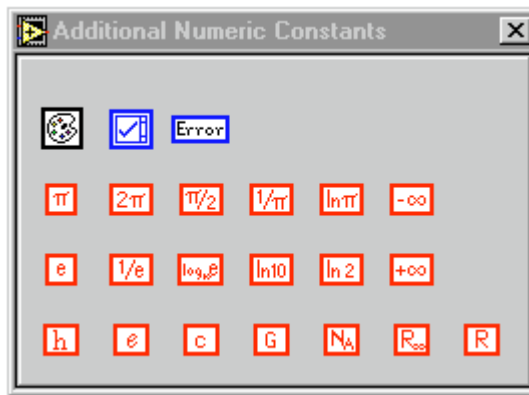
Re/Im To Complex

Creates a complex number from two values in rectangular notation.



Additional Numeric Constants Descriptions

The following illustration shows 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 to the Item Symbols attribute.



Color Box Constant

Use this constant to supply a constant color value to the block diagram. Set this value by clicking 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 **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.

1/e **Avogadro Constant (1/mol)**

Returns the value $6.0220e23$.

ln 2 **Base 10 Logarithm of e**

Returns the value 0.43429448190325183 .

c **Elementary Charge (c)**

Returns the value $1.6021892e-19$.

G **Gravitational Constant (Nm²/kg²)**

Returns the value $6.6720e-11$.

R **Molar Gas Constant (J/mol K)**

Returns the value 8.31441 .

e

Returns the value $2.7182818284590452e+0$.

ln π **Natural Logarithm of Pi**

Returns the value 1.14472988584940020 .

ln 2 **Natural Logarithm of 2**

Returns the value 0.69314718055994531 .

ln 10 **Natural Logarithm of 10**

Returns the value 2.30234095236904570 .

-∞ **Negative Infinity**

Returns the value $-\infty$.

**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 $1.097373177e7$.

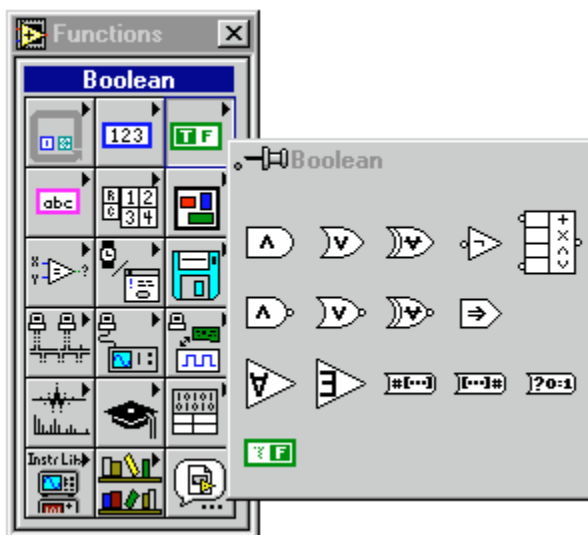
**Speed of Light in Vacuum (m/sec)**

Returns the value 299,792,458.

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 a long integer.

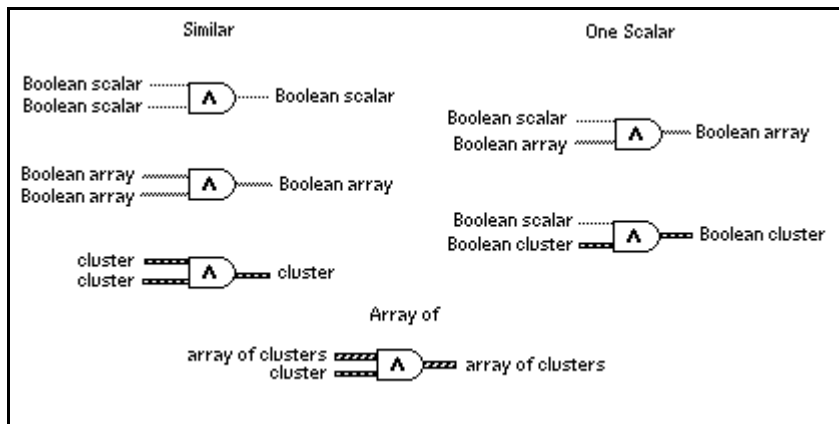
The logical functions work on arrays of numbers or Boolean values, clusters of numbers or Boolean values, arrays of clusters of numbers or Boolean values, and so on.

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.

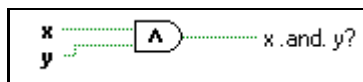


Boolean Function Descriptions

The following Boolean functions are available.

And

Computes the logical AND of the inputs.



Note

This function performs bit-wise operations on numeric inputs.

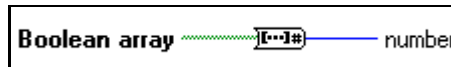
And Array Elements

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



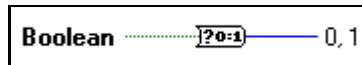
Boolean Array To Number

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.



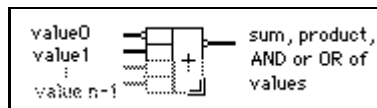
Boolean To (0,1)

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

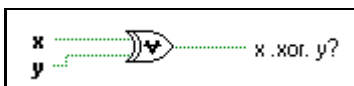
You can invert the inputs or the output of this function by popping up on the individual terminals and selecting **Invert**. For Add, select **Invert** to negate an input or the output. For Multiply, select **Invert** to use the reciprocal of an input or to produce the reciprocal of the output. For AND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

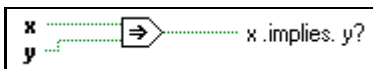
Exclusive Or

Computes the logical exclusive OR of the inputs.



Implies

Computes the logical OR of y and of the logical negation of x . The function negates x then computes the logical OR of y and of the negated x .



Not

Computes the logical negation of the input.



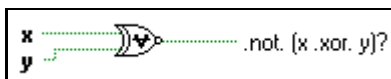
Not And

Computes the logical NAND of the inputs.



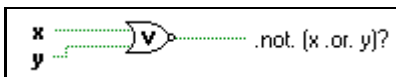
Not Exclusive Or

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



Not Or

Computes the logical NOR of the inputs.



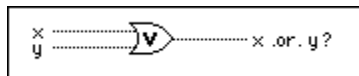
Number To Boolean Array

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.



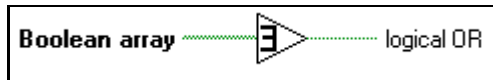
Or

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

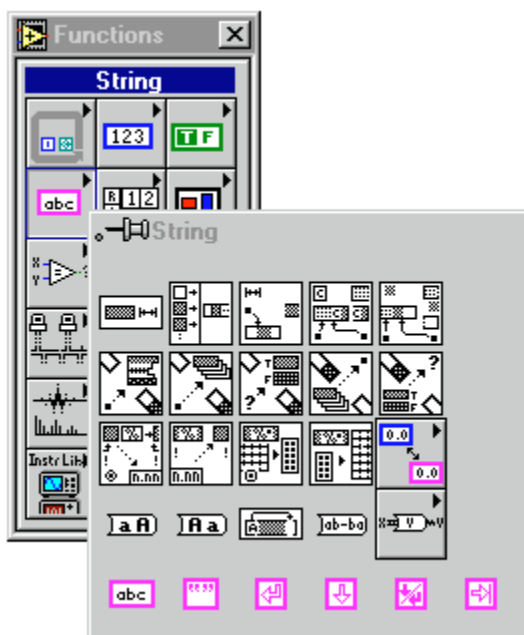


Use this function to supply a constant TRUE/FALSE value to the block diagram. Set this value by clicking the **T** or **F** portion of the constant with the Operating tool. This value cannot be changed while the VI executes. You can assign a label to this constant.

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 C, *GPIB Multiline Interface Messages* 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. 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 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 produce a string as output, 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)	<p>When scanning, specifies an exact field width to use. G scans only the specified number of characters when processing the parameter.</p> <p>When formatting, specifies the minimum character field width of the output. This width 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.</p>
.	Separates width from Precision.
Precision (optional)	<p>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.</p> <p>For string parameters, specifies the maximum width of the field. G truncates strings longer than this length.</p>

Table 6-2. String Syntax (Continued)

Syntax Element	Description
{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	<p>Single character that specifies how to scan or format perimeter, as follows:</p> <ul style="list-style-type: none"> d decimal integer x hex integer o octal integer b binary integer f floating-point number with fractional format e floating-point number with scientific notation g floating-point number using e format if the exponential is less than -4 or greater than Precision, or f format otherwise s string <p>An l (lowercase L) preceding the conversion</p>
Localization Codes	<p>Codes used as format separators for localization, as follows:</p> <ul style="list-style-type: none"> %, ; comma decimal separator %. ; period decimal separator %; system default separator

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, the datatype of the parameter determines the size of an integer and whether the integer is signed or unsigned.

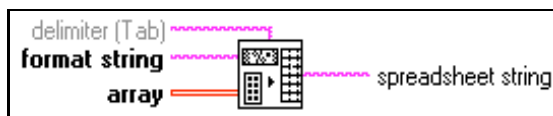
For examples of format string usage, see the Format Into String and Scan From String function descriptions later in this chapter.

String Function Descriptions

The following string functions are available.

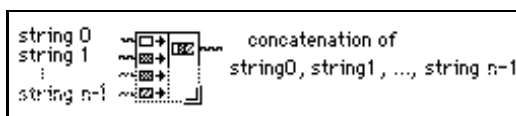
Array To Spreadsheet String

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, separated pages.



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, then stretching it until you reach the desired number of arguments.

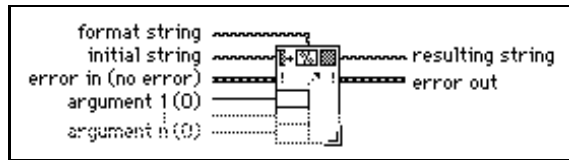


Table 6-3 shows the errors that can appear in **error out** by the Format Into String function.

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. You must correct these errors 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.

Table 6-4. Format Specifiers (Continued)

Format String	Argument(s)	Resulting String
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.

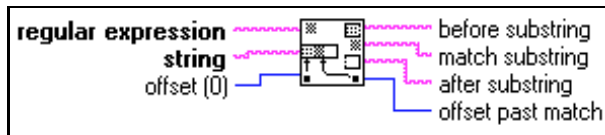


Table 6-5. Special Characters for Match Pattern

Special Character	Interpreted by the Match Pattern Function as...
.	Matches any character.
?	Matches zero or one instances of the expression preceding ?.
\	<p>Cancels the interpretation of special characters (for example, \? matches a question mark). You can also use the following constructions for the space and non-displayable characters:</p> <p>\b backspace</p> <p>\f form feed</p> <p>\n newline</p> <p>\s space</p> <p>\r carriage return</p> <p>\xx any character, where xx is the hex code using 0 through 9 and upper case A through F</p> <p>\t tab</p>
^	<p>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.</p>
[]	<p>Encloses alternates. For example, [abc] matches a, b, or c. The following character has special significance when used within the brackets:</p> <p>- (dash) Indicates a range when used between digits, or lowercase or uppercase letters (for example, [0-5],[a-g], or [L-Q])</p> <p>The following characters have significance only when they are the first character within the brackets:</p> <p>~ Excludes the set of characters, including nondisplayable characters. [~0-9] matches any character other than 0 through 9.</p> <p>^ 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.</p>

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

Special Character	Interpreted by the Match Pattern Function as...
+	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

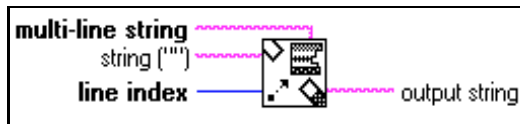
Characters to Be Matched	Regular Expression
VOLTS	VOLTS
All uppercase and lowercase versions of volts, that is, VOLTS, Volts, volts, and so on	[Vv][Oo][Ll][Tt][Ss]
A space, a plus sign, or a minus sign	[+-]
A sequence of one or more digits	[0-9]+
Zero or more Spaces	\s* or * (that is, a space followed by an asterisk)
One or more Spaces, Tabs, New Lines, 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	(.*)

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

Characters to Be Matched	Regular Expression
The longest string within parentheses but not containing any parentheses within it	<code>([~ ()] *)</code>
The character [<code>[[]</code>

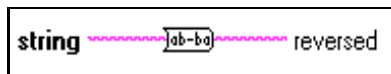
Pick Line & Append

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



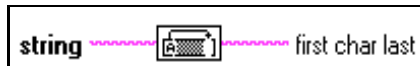
Reverse String

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, 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.

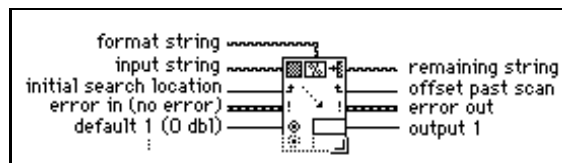


Table 6-7 lists the Scan from String errors.

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.
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 the Scan-failed error can occur at run time.

Table 6-8 lists Scan From String examples.

Table 6-8. Scan from String Examples

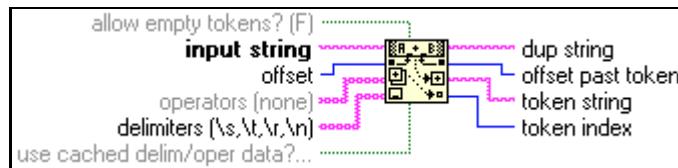
Input String	Format String	Default(s)	Output(s)	Remaining String
abc xyz 12.3+56i 7200	%s %s%f%2d	— — 0&0i (CDB) —	abc xyz 12.3+56i 72	00
Q+1.27E-3 tail	Q%f t	—	1.27E-3	ail

Table 6-8. Scan from String Examples (Continued)

Input String	Format String	Default(s)	Output(s)	Remaining String
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

Scan String for Tokens

Scans **input string**, starting at **offset**, and returns the next token found.

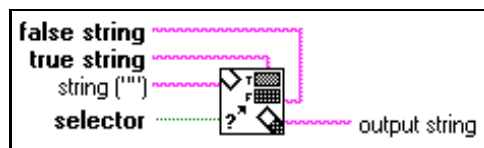


A *token* is a substring of **input string**, which is surrounded by **delimiters**, or which matches an element in **operators**. Typically, tokens represent individual keywords, numeric values, or **operators** found when parsing a configuration file or other text-based data format. This function scans **input string**, starting at **offset**, returning the next token found.

See the online reference for more information about the Scan String for Tokens function and parameters.

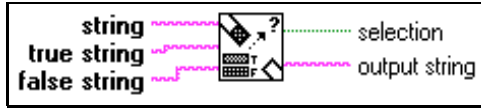
Select & Append

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



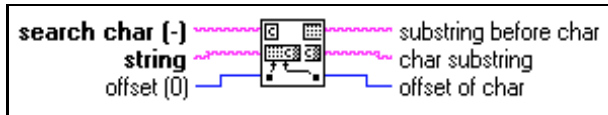
Select & Strip

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 **string**, beginning at **offset**, and splits the string at that point.



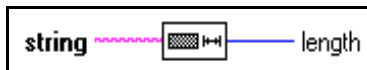
Spreadsheet String To Array

Converts **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.



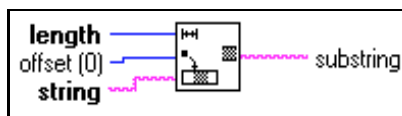
String Length

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



String Subset

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



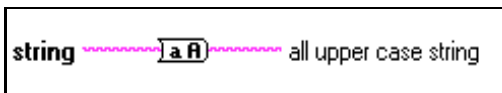
To Lower Case

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



To Upper Case

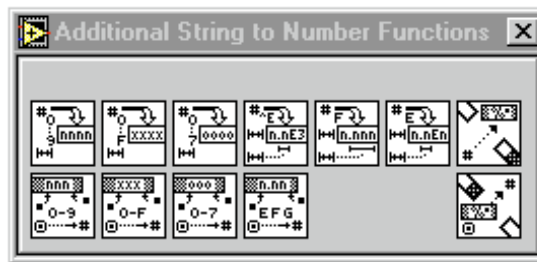
Converts all alphabetic characters in **string** to uppercase characters. This function does not affect non-alphabetic characters.



Additional String To Number Function Descriptions

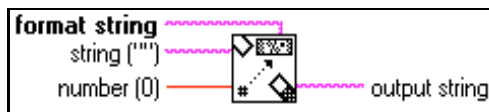
For general information about Additional String to Number functions, see *Polymorphism for Additional String to Number Functions*, earlier in this chapter.

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**.

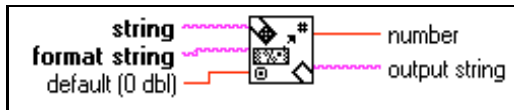




Note *The **Format Into String** function has the same functionality as **Format & Append** but can use multiple inputs, so you can convert information simultaneously. Consider using **Format Into String** instead of this function to simplify your block diagram.*

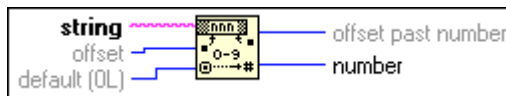
Format & Strip

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**.



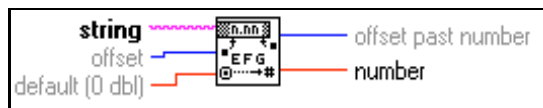
From Decimal

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



From Exponential/Fract/Eng

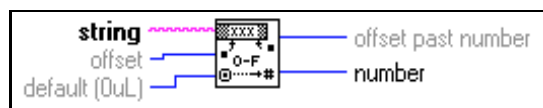
Interprets the characters 0 through 9, plus, minus, e, E, and the decimal point (usually period) in **string** starting at **offset** as a floating-point number in engineering notation, or exponential or fractional format and returns it in **number**.



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

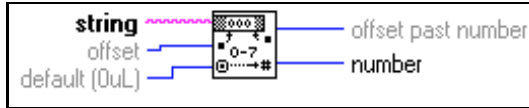
From Hexadecimal

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**.



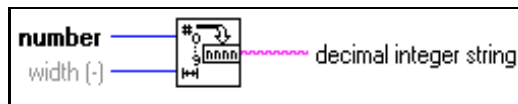
From Octal

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.



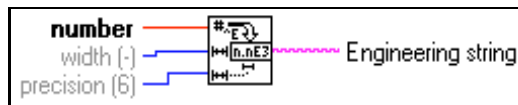
To Decimal

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).



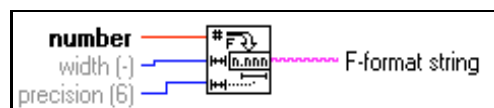
To Exponential

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



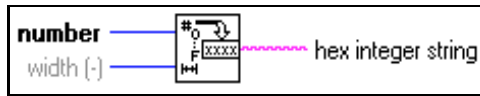
To Fractional

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



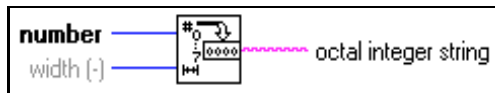
To Hexadecimal

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.

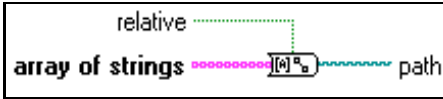


Array Of Strings To Path accepts one-dimensional (1D) arrays of strings, Path To Array Of Strings accepts paths, Path To String accepts paths, and String To Path accepts strings.

Array Of Strings To Path

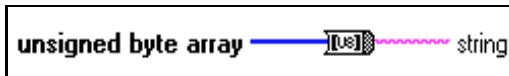
Converts **array of strings** into a relative or absolute **path**.

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 change as moving up a level in directory hierarchy.



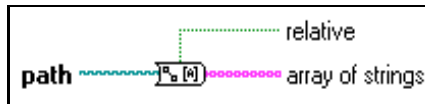
Byte Array To String

Converts an array of unsigned bytes into a string.



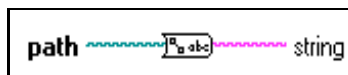
Path To Array Of Strings

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



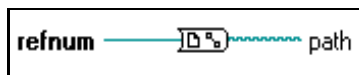
Path To String

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



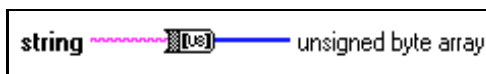
Refnum To Path

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



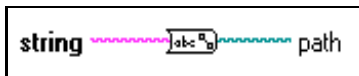
String To Byte Array

Converts **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 constant to supply a constant ASCII string to the block diagram. Set this string by clicking the constant with the Operating tool and typing the value. You can change the display mode so you can see non-displayable characters or the hex equivalent to the characters. You also can set the constant in password display mode so asterisks are displayed when you type 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 for UNIX, the value is LF.

Line Feed



Consists of a constant string containing the ASCII LF value.

Tab

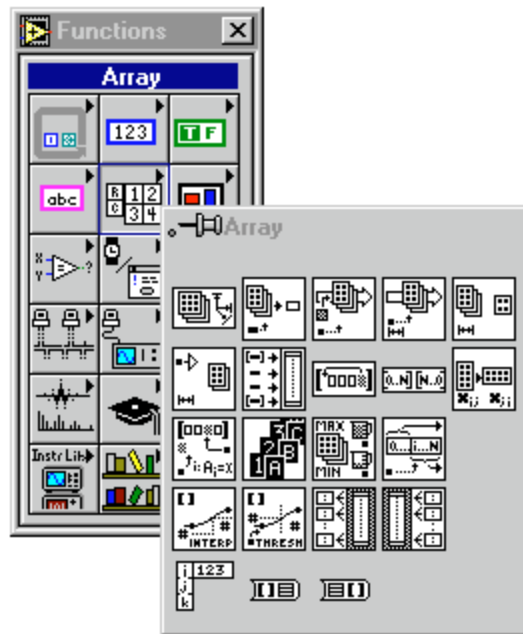


Consists of a constant string containing the ASCII HT (horizontal tab) value.

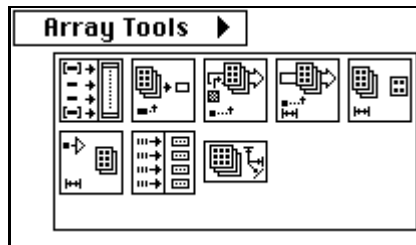
Array Functions

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



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

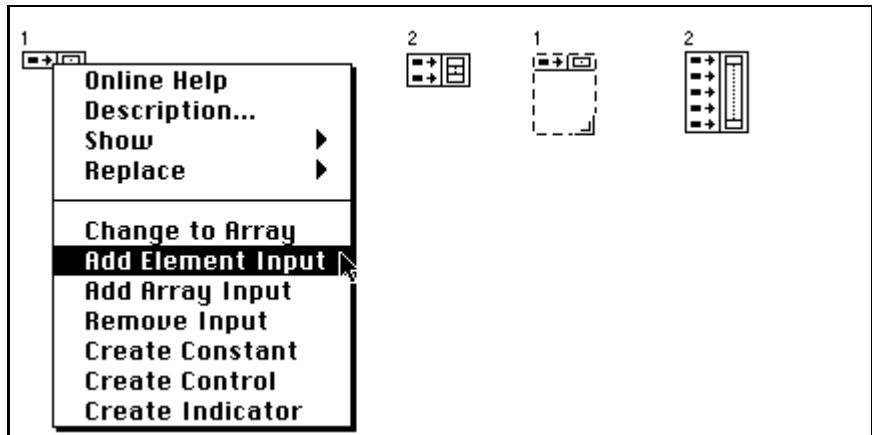
For examples of array functions, see `examples\general\arrays.llb`.

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 **Add Element Input** or **Add Array Input** and **Remove Input** pop-up menu 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 pointer 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

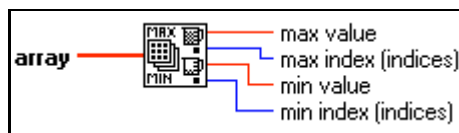
Most of the array functions accept n -dimensional arrays of any type. However, the wiring diagrams in the function descriptions show numeric arrays as the default data type.

Array Function Descriptions

The following Array functions are available.

Array Max & Min

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

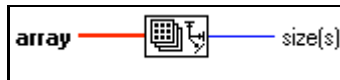


If a numeric **array** has one dimension, the **max index** and **min index** outputs are scalar integers. If a numeric **array** has more than one dimension, these outputs are 1D arrays that contain the indices of the maximum and minimum values.

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

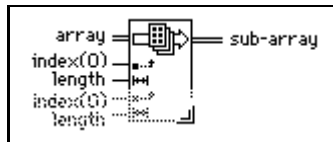
Array Size

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

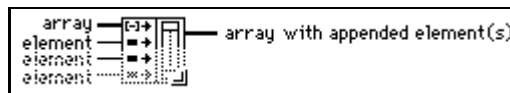
Converts a 1D array to a cluster of elements of the same type as the array elements. Pop up on the node to set the number of elements in the cluster. The default is nine. The maximum cluster size for this function is 256.



For more information about 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

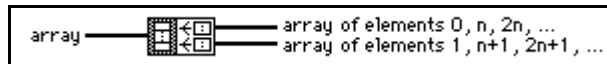
Converts a cluster of identically typed components to a 1D array of elements of the same type.



For more information about clusters, see Chapter 8, [Cluster Functions](#).

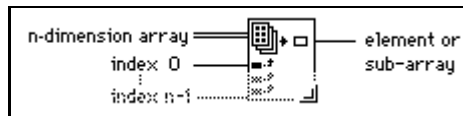
Decimate 1D Array

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



Index Array

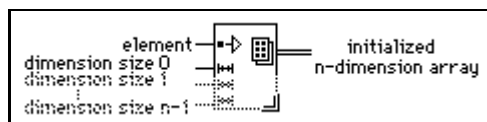
Returns the **element** of **array** at **index**. If **array** is multidimensional, you must add additional **index** terminals for each dimension of **array**.



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.

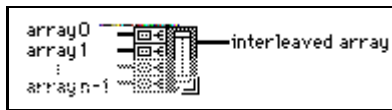
Initialize Array

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



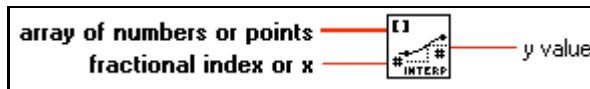
Interleave 1D Arrays

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



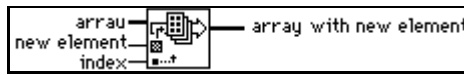
Interpolate 1D Array

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



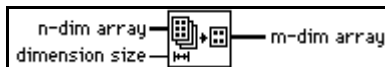
Replace Array Element

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



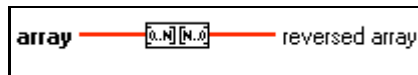
Reshape Array

Changes the dimension of an array according to the value of **dimension size**. The function is resizable; **m-dim array** has one dimension for each **dimension size** input. For example, you can use this function to change a 1D array into a 2D array or vice versa. You also can use it to increase and decrease the size of a 1D array.



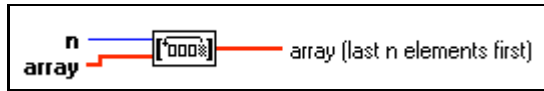
Reverse 1D Array

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



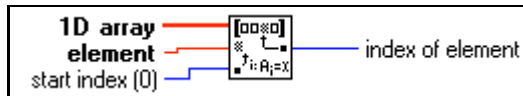
Rotate 1D Array

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



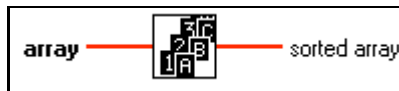
Search 1D Array

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



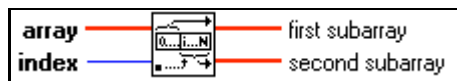
Sort 1D Array

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

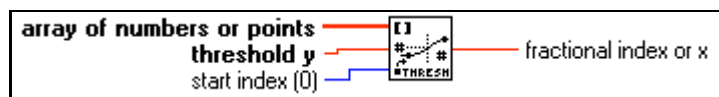
Divides **array** at **index** and returns the two portions.



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.

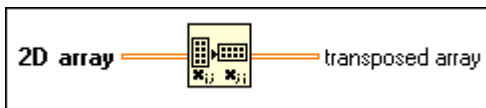


For example, suppose **array of numbers or points** is an array of four numbers [4, 5, 5, 6], **start index** is 0, and **threshold y** is 5. The **fractional index or x** is 1, corresponding to the index of the first value of 5 the function finds. Suppose the array elements are 6, 5, 5, 7, 6, 6, the **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

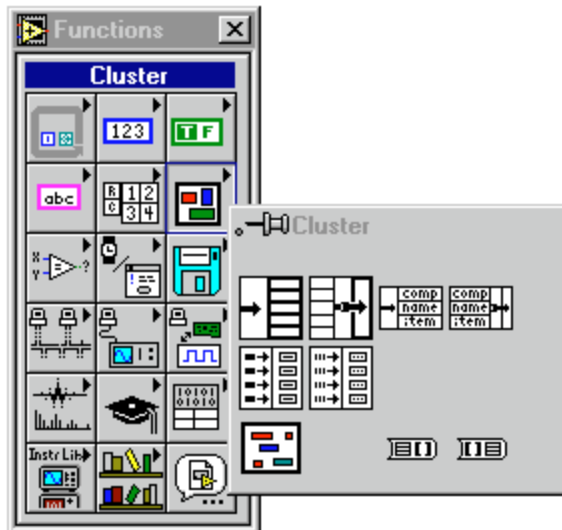
Rearranges the elements of **2D array** such that **2D array**[i,j] becomes **transposed array**[j,i].



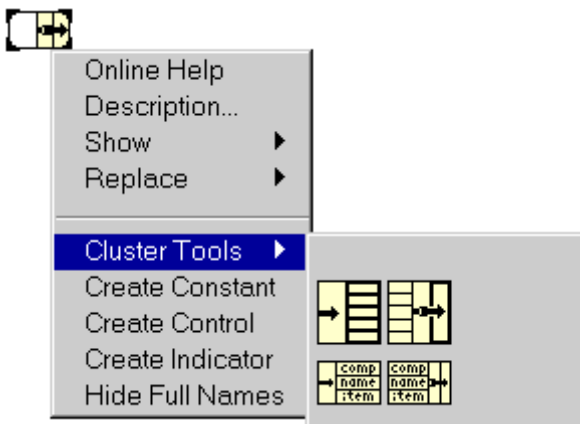
Cluster Functions

This chapter describes the functions for cluster operations.

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



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



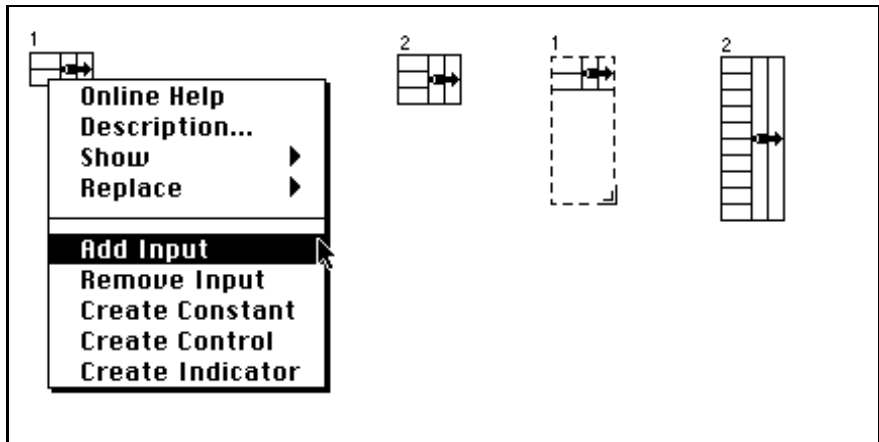
If you select the functions from this palette, they appear with the correct number of terminals to wire to the object on which you popped up.

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 **Add Input** or **Remove Input** pop-up menu options or by resizing the node using the Positioning tool. If you want to add terminals by popping up, 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 an element with the cluster order cursor sets the place of the element 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 the **Enter** button to set it and exit the cluster order edit mode. Click 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.

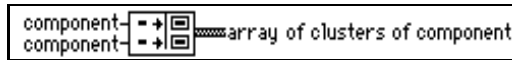
Array To Cluster

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.



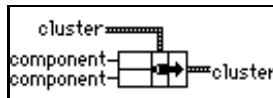
Build Cluster Array

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.



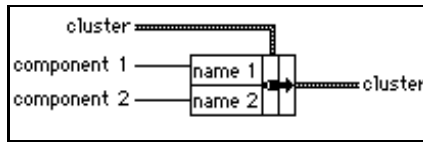
Bundle

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.

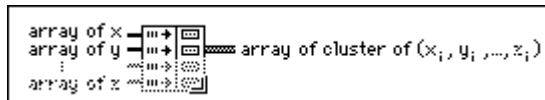
Cluster To Array

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

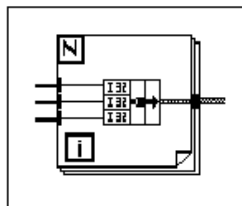


Index & Bundle Cluster Array

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.

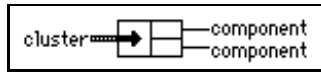


This function is equivalent to the following block diagram and is useful for converting a cluster of arrays to an array of clusters.



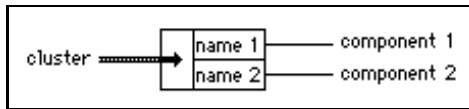
Unbundle

Disassembles a cluster into its individual components.



Unbundle By Name

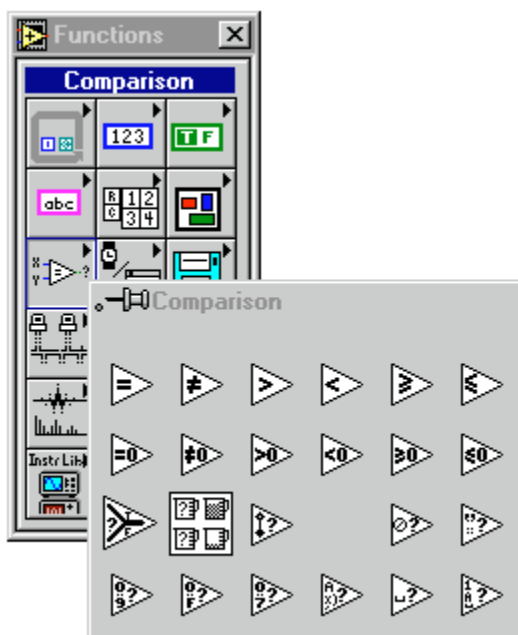
Returns the cluster elements whose names you specify. You select the element you want to access by popping up on the name output terminals and selecting a name from the list of elements in the cluster.



Comparison Functions

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

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



Comparison Function Overview

This section introduces the Comparison functions.

Boolean Comparison

The Comparison functions treat the Boolean value TRUE as greater than the Boolean value FALSE.

String Comparison

These functions compare strings according to the numerical equivalent of the ASCII characters. Therefore, 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 compares the strings `abcd` and `abef` until it finds `c`, which has a value less than the value of `e`. The presence of a character is greater than the absence of one. Therefore, the string `abcd` is greater than `abc` because the first string is longer.

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.

Numeric Comparison

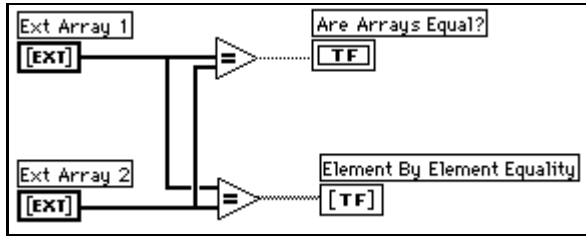
Most of the Comparison functions test one input or compare two inputs and return a Boolean value. The functions convert numbers to the same representation before comparing them. Comparisons with a value of not a number (NaN) return a value that indicates inequality.

Cluster Comparison

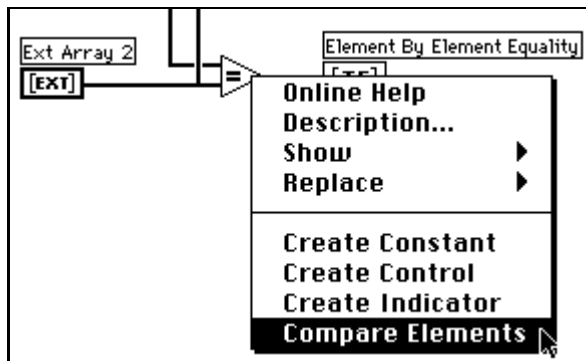
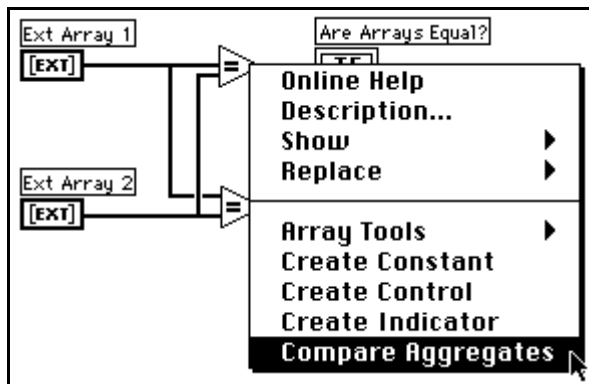
The Comparison functions compare clusters the same way they compare strings, one element at a time starting with the 0th element until an inequality occurs. Clusters must have the same number of elements, of the same type, and in the same order if you want to compare them.

Comparison 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. 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. Therefore, 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.

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

For clusters using the **Compare Aggregates** mode, LabVIEW compares using cluster order. The two clusters LabVIEW compares must have the same number of elements.

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 the type of a character. 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 C, *GPIB Multiline Interface Messages*, for the numbers that correspond to each ASCII 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 release Boolean values as output 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 the same data structure as the input. See Chapter 11, *File Functions*, and Chapter 31, *Introduction to LabVIEW Instrument I/O VIs*, for more information about 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.

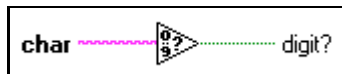
Comparison functions that use arrays and clusters normally produce Boolean arrays and clusters of the same structure. You can pop-up and change to **Compare Aggregates**, in which case the function releases a single Boolean value as output. The function compares aggregates by comparing the first set 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.

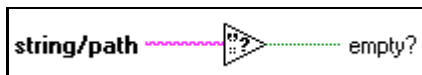
Decimal Digit?

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



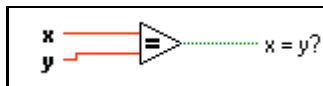
Empty String/Path?

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.



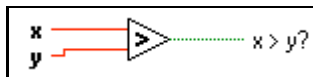
Equal To 0?

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



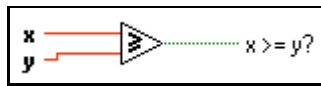
Greater?

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



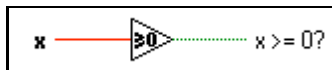
Greater Or Equal?

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



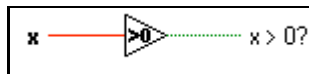
Greater Or Equal To 0?

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.



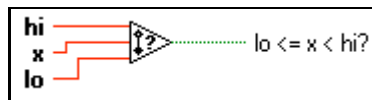
Hex Digit?

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.



In Range?

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

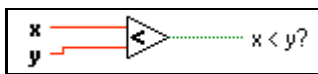


Note

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.

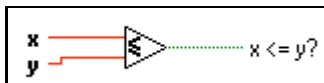
Less?

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



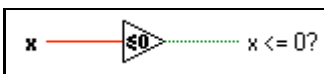
Less Or Equal?

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



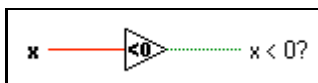
Less Or Equal To 0?

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 **class number** for **char**.

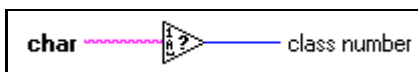


Table 9-1. Lexical Class Number Descriptions

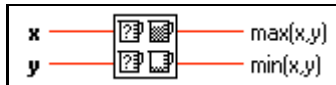
Class Number	Lexical Class
0	Extended characters with a Command- or Option- key prefix (codes 128 through 255)
1	Non-displayable ASCII characters (codes 0 to 31 excluding 9 through 13)
2	White space characters: Space, Tab, Carriage Return, Form Feed, Newline, and Vertical Tab (codes 32, 9, 13, 12, 10, and 11, respectively)

Table 9-1. Lexical Class Number Descriptions (Continued)

Class Number	Lexical Class
3	Digits 0 through 9
4	Uppercase characters A through Z
5	Lowercase characters a through z
6	All printable ASCII non-alphanumeric characters

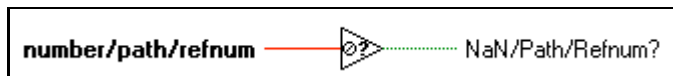
Max & Min

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



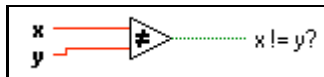
Not A Number/Path/Refnum?

Returns TRUE if **number/path/refnum** is not a number (NaN), not a path, or not a refnum. Otherwise, this function returns FALSE. NaN can be the result of dividing by 0, calculating 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.



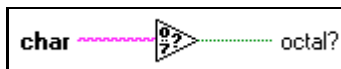
Not Equal To 0?

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



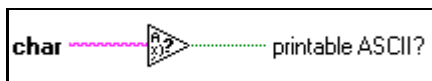
Octal Digit?

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.



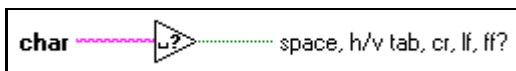
Select

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?

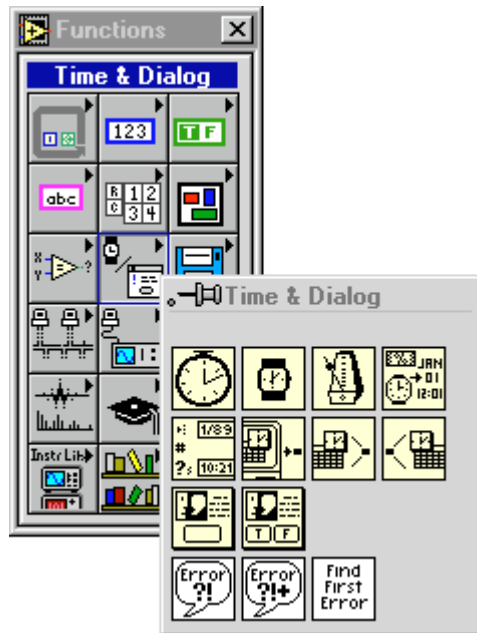
Returns TRUE if **char** is a white space character, such as Space, Tab, Newline, Carriage Return, Form Feed, or Vertical Tab. Otherwise, the function returns FALSE.



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 that you access by selecting **Functions»Time & Dialog**.



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. Valid Value of Elements for Date/Time Cluster

	Element	Valid Values
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. Therefore, they do not complete execution until the specified time has elapsed. The functions use asynchronous calls, so other nodes can execute while the timing nodes wait.



Note

Time values outside 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 might not convert to the same date on all platforms. This exception primarily exists on Windows 3.x, which does not support dates prior to Jan. 1, 1970, Universal Time.

Error Handling Overview

Every time you design a program, consider the possibility that something can go wrong and, if it does, you should consider how your program can manage the problem. LabVIEW automatically notifies you with a dialog box only when a few run-time errors occur, mostly for file-dialog operations. LabVIEW does not report all errors. If it reported 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 type of 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, you only know 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 might be powered off, broken down, or otherwise unable to complete its normal tasks. 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 might have felt safe ignoring, such as those from functions that close files or clear DAQ operations. Be sure to check all I/O operations for errors.

It might seem easier to ignore error checking when you must 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.

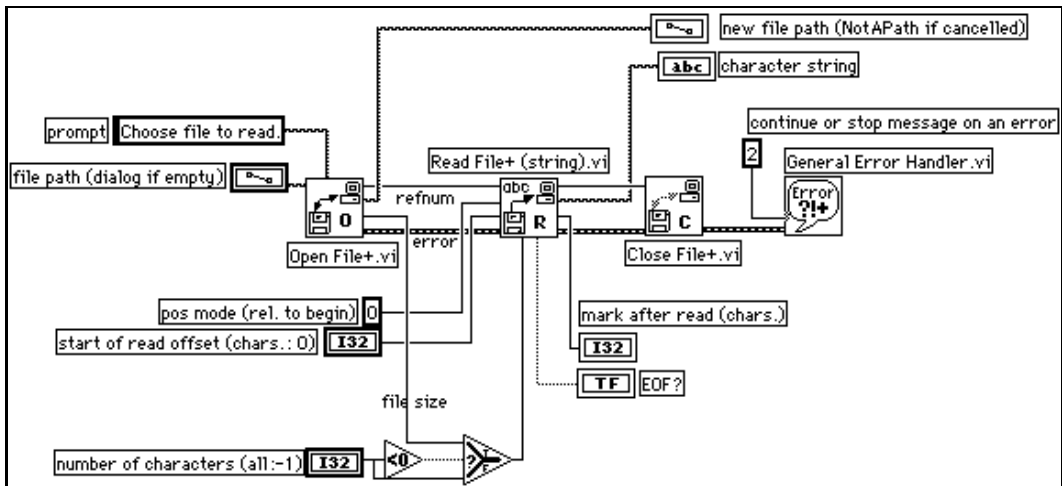
G functions and library VIs return errors in one of two ways—with numeric error codes or with an error state cluster. Typically, functions release output error codes while VIs incorporate the error cluster, usually within a framework called error input/output (error I/O).

Error I/O and the Error State Cluster

The concept of error I/O is logical for the G dataflow architecture. If data information can flow from one node to another, so can error state information. Each node that needs information 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, 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.vi, shows how error I/O is implemented in a simple VI.

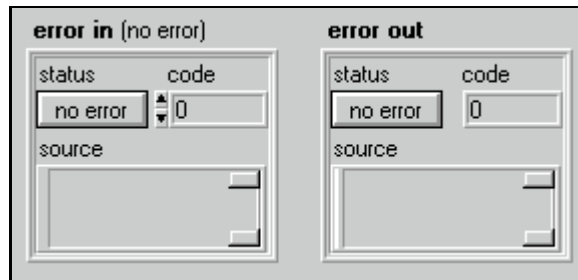


The operation starts at `Open File+.vi`. If it opens the file successfully, `Read File+(string).vi` reads the file and `Close File+.vi` closes the file. If you pass in an invalid path, `Open File+.vi` detects the error and passes the error state through the other two VIs to the General Error Handler, 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.

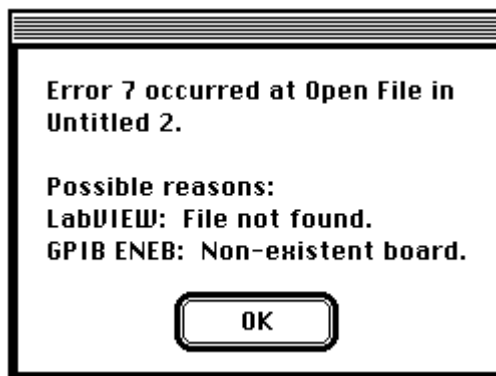
The error state consists of three pieces of information that are combined into the error cluster. The **status** is a Boolean value—TRUE if an error exists, FALSE if it does not. The **code** consists of a signed 32-bit integer that identifies the error. A non-zero error **code** coupled with a FALSE error **status** signals a warning rather than a fatal error. For example, a DAQ timeout event (code 10800) typically is 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 an invalid path.



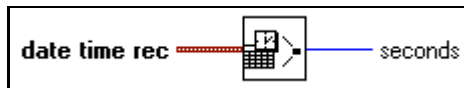
The General Error Handler is one of the three error-handling utility VIs. It contains a database of error codes and descriptions, from which it creates messages like the previous one. The Simple Error Handler performs the same basic operation but has fewer options. The third VI, Find First Error, creates 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.

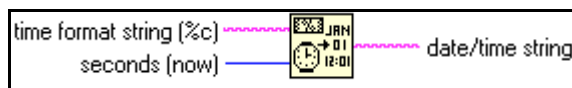


The day of week, day of year, and DST integers are ignored. If any of the other integers are out of the ranges specified in Table 10-1, the results are unpredictable.

When used as an integer, the day of month integer has a valid range of 1 to 366. 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.

Format Date/Time String Function

Gives you the ability to display the date and time in a format you specify.



The **date/time string** is determined from the **seconds (now)**, which is the number of seconds since 12:00 a.m., January 1, 1904, Universal Time, and **time format string** is the format of the output string.

If **seconds** is not wired, the current time is used. If **time format string** is not wired, the default is %c, which corresponds to the date/time representation appropriate for the current locale.

The Format Date/Time String function calculates **date/time string** by copying **time format string** and replacing each of the format codes with the corresponding values in the following table.

Table 10-2. Format Codes for the Time Format String


Format Code	Value
%%	a single percent character
%a	abbreviated weekday name (e.g. Wed)
%A	full weekday name (e.g. Wednesday)
%b	abbreviated month name (e.g. Jun)
%B	full month name (e.g. June)
%c	locale's default date and time representation
%d	day of month (01–31)
%H	hour (24-hour clock) (00–23)
%I	hour (12-hour clock) (01–12)
%j	day number of year (001–366)
%m	month number (01–12)
%M	minute (00–59)
%p	AM or PM flag
%S	seconds (00–59)
%U	week number of the year (00–53), with Sunday as the first day of the week
%w	weekday as a decimal number (0–6), with 0 representing Sunday
%W	week number of the year (00–53), with Monday as the first day of the week
%x	date representation of locale
%X	time representation of locale
%y	year within century (00–99)
%Y	year, including the century (for example, 1997)
%Z	time zone name or abbreviation

Characters appearing in **time format string** that are not part of a format code are copied to the output string verbatim. Time format codes (beginning with %) that are not recognized output the character literally.

Time format codes always have leading zeros as necessary to ensure a constant field width. An optional # modifier before the format code letter removes the leading zeros from the following format codes:

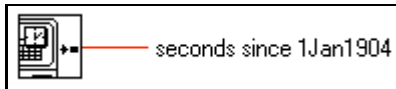
%#d, %#H, %#I, %#j, %#m, %#M, %#S, %#S, %#U, %#w, %#W, %X, %#y, %#Y.

The # modifier does not modify the behavior of any other format codes.

 **Note** *The %c, %x, %X, and %Z format codes depend on operating system locale support; the output of these codes is platform dependent. Interpretation of the Daylight Savings Time rule also can vary per platform.*

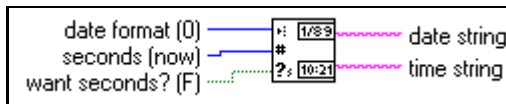
Get Date/Time In Seconds

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



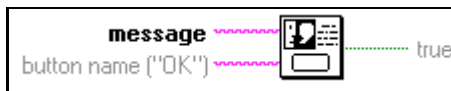
Get Date/Time String

Converts a time-zone independent number calculated 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.



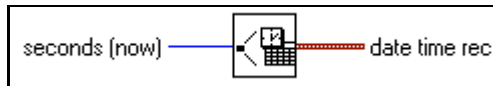
One Button Dialog Box

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



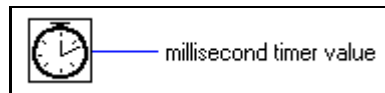
Seconds To Date/Time

Converts a time-zone-independent number calculated 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 the local time (second, minute, hour, day of the month, month, year, day of the week, day of the year, and Standard or Daylight Savings Time) in the configured time zone for your computer. The Standard or Daylight Savings time parameter is set according to the operating system setting for Daylight Savings and indicates whether the date/time cluster was adjusted due to Daylight Savings Time.



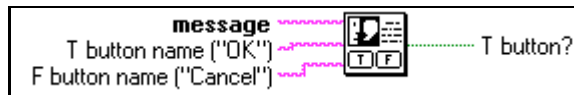
Tick Count (ms)

Returns the value of the millisecond timer. The base reference time (millisecond zero) is undefined; therefore, 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.



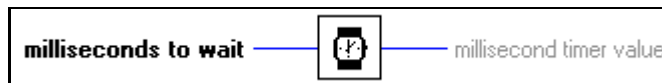
Two Button Dialog Box

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.



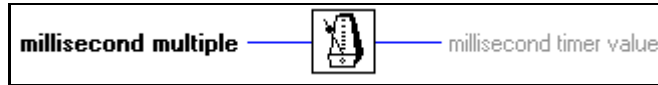
Wait (ms)

Waits the specified number of milliseconds 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**. 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 might be short.

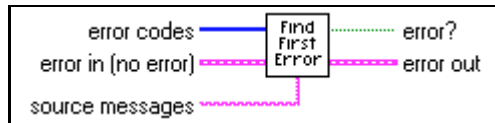


Error Handling VI Descriptions

The following Error Handling VIs are available.

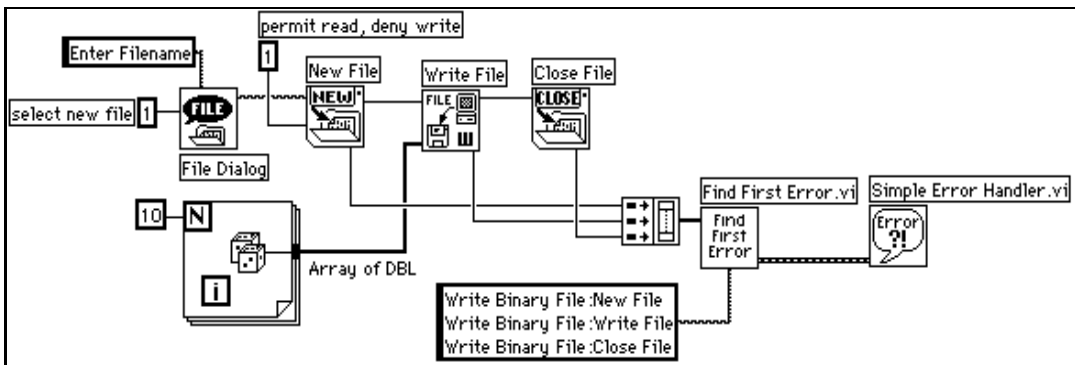
Find First Error

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



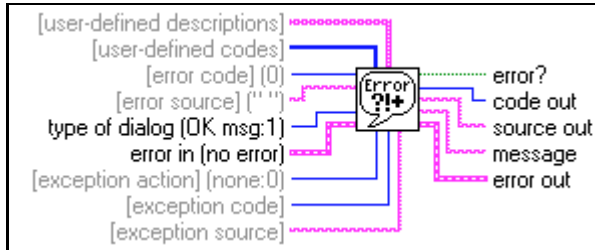
If this VI finds an error, it sets the parameters in the **error out** cluster. You can wire this cluster to the Simple or General Error Handler to identify the error and describe it to the user.

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.



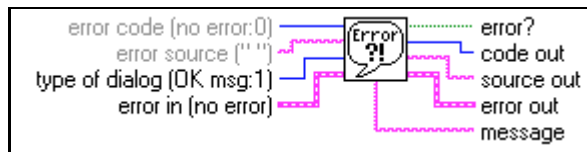
General Error Handler

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



Simple Error Handler

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

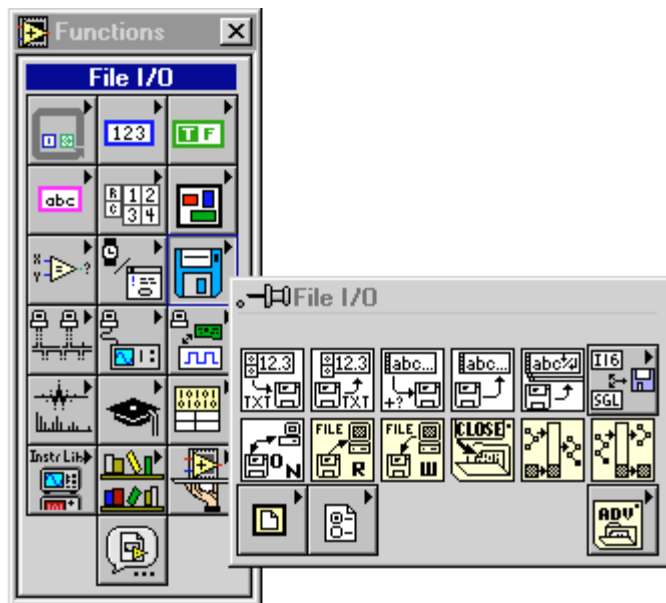


Simple Error Handler calls General Error Handler and has the same basic functionality as General Error Handler, but with fewer options.

File Functions

This chapter describes the low-level VIs and functions that manipulate files, directories, and paths. This chapter 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
- Configuration 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 File 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 numbers or signed word integers to byte stream files

The high-level file VIs described here call the low-level file 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:

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

Low-Level File VIs and File Functions

The low-level file VIs and 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 to be manipulated. The file refnum is an identifier associated 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.

Refer to the *LabVIEW Online Tutorial: Introduction to LabVIEW* 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 the G datatypes. Byte stream files can be created using high-level File VIs and low-level File VIs and 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.

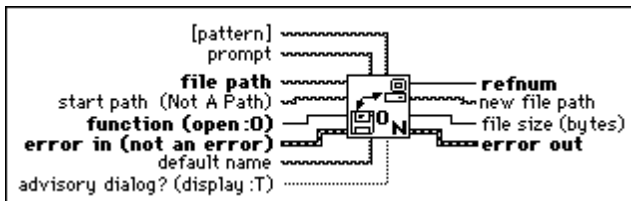
Flow-Through Parameters

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.

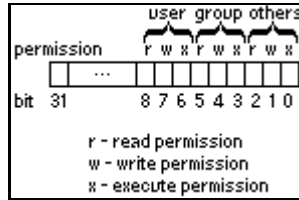
Permissions

Some of the file functions have a 32-bit integer parameter called **permissions** or **new permissions**. These functions use only the least significant nine bits of the 32-bit integer to determine file and directory access permissions.

(Windows) The permissions are ignored 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) All 9 bits of permissions are used for directories. The bits that 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 the 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.

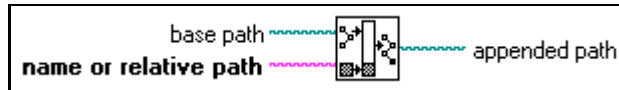


File I/O Function and VI Descriptions

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.

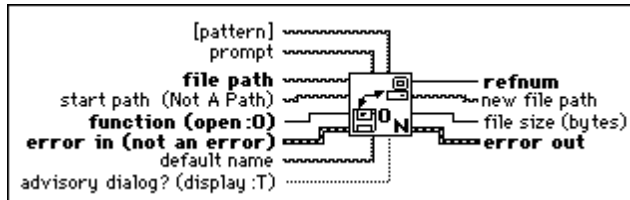


Note

The Close File VI handles error I/O differently than other file functions; it executes even when its error in indicates that an error has occurred in a preceding function.

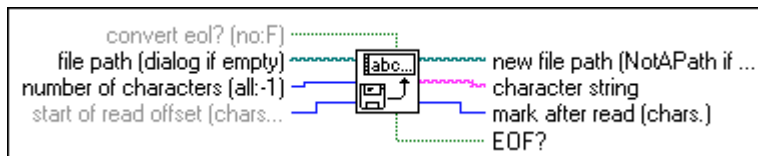
Open/Create/Replace File

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 Write File or Read File functions.



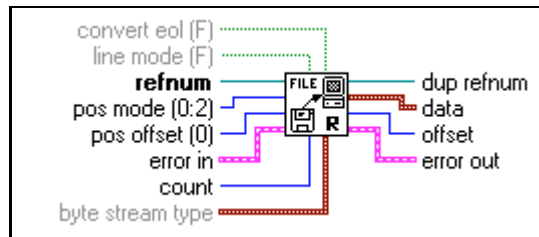
Read Characters From File

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.



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

If you wire **count**, it can be a scalar number, in which case the function returns a 1D array of instances of **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 **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 and **count** is 3, 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 10 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.)

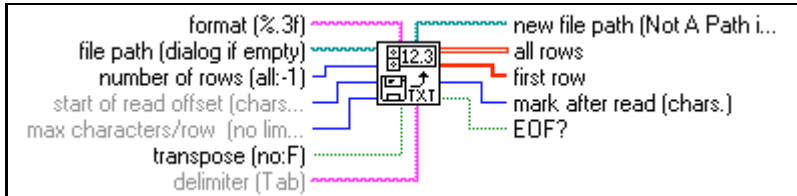
If you wire **count**, it can be a scalar number, in which case the function returns a 1D array of records. Or it can be a cluster of N scalar numbers, in which case the function returns an N-dimension array of records.

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**.

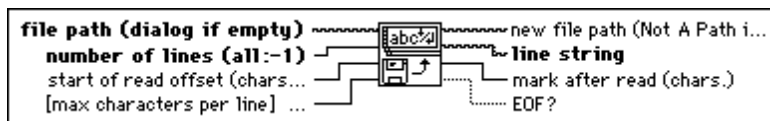
Read From Spreadsheet File

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.



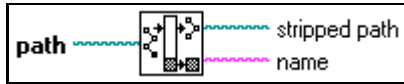
Read Lines From File

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.



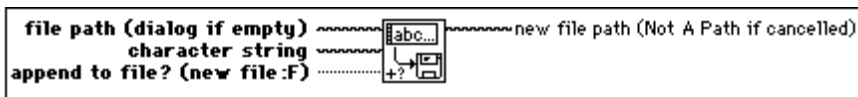
Strip Path

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



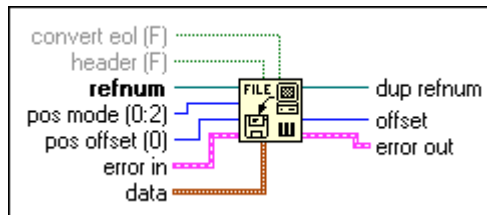
Write Characters To File

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.



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. Write File 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, Write File ignores **convert eol**.

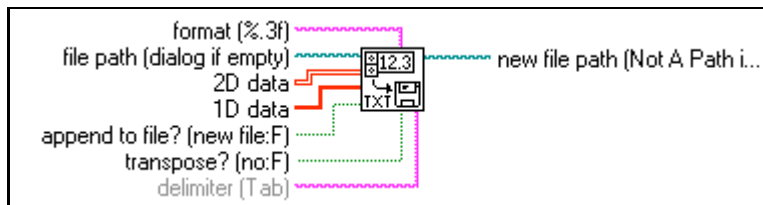
Writing Datalog Files

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). Write File 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.

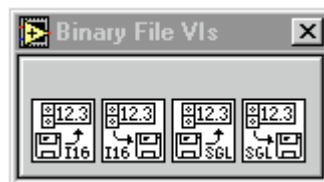
Write To Spreadsheet File

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.



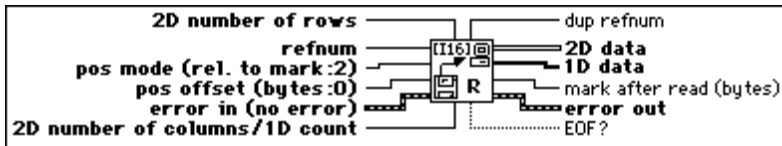
Binary File VI Descriptions

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



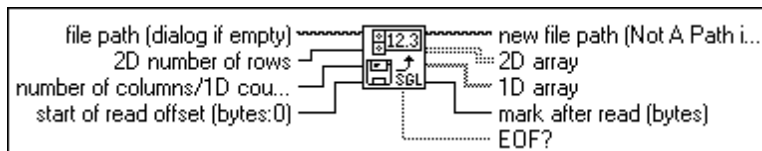
Read From I16 File

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.



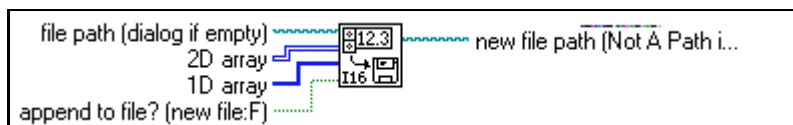
Read From SGL File

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.



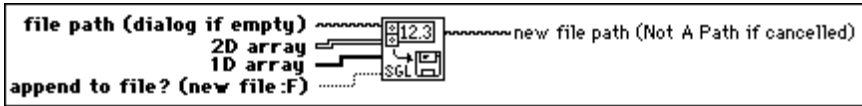
Write To I16 File

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.



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.



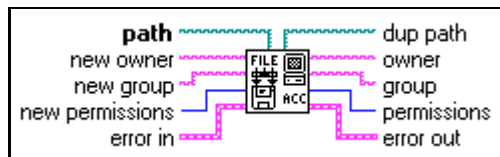
Advanced File Function Descriptions

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.

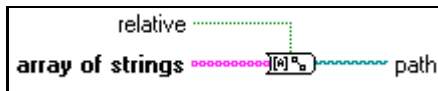


(Windows) The Access Rights function ignores **new owner** and **new group** and returns empty strings for **owner** and **group** because Windows does not support owners and groups.

(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 **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.



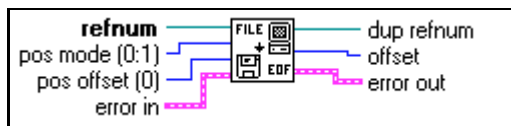
Delete

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

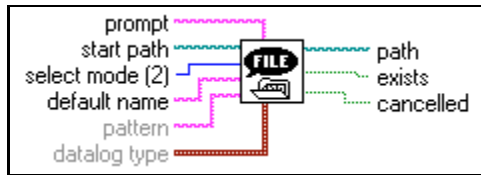
Sets and returns the logical EOF (end-of-file) of the file identified by **refnum**. **pos mode** and **pos offset** specify the new location of the EOF. If you do not specify **pos mode** or **pos offset**, this function returns the current unchanged EOF. This function always returns the location of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

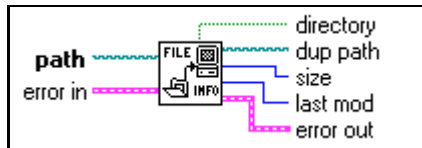
File Dialog

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.



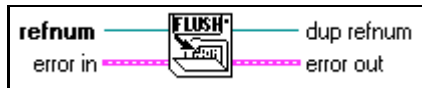
File/Directory Info

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.

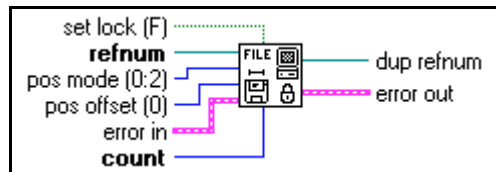
List Directory

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

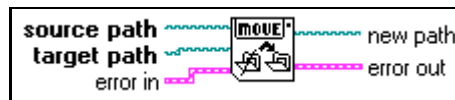
Locks or unlocks a range of a file specified by **refnum**. Locking a range of a file prevents both reading and writing by other users, overriding permissions for the file, and the deny mode associated with **refnum**. See the *File I/O VI and Function Overview* section in this chapter for a full discussion 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.

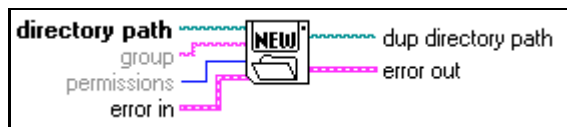
Move

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



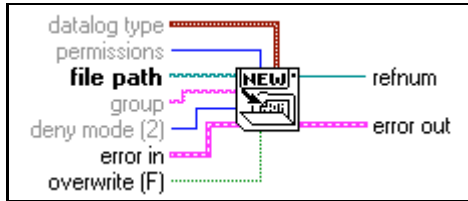
New Directory

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.



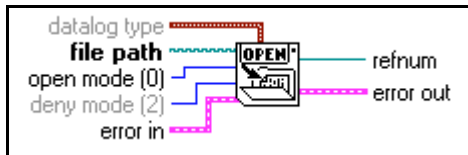
New File

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



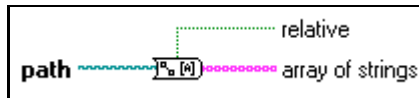
Open File

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



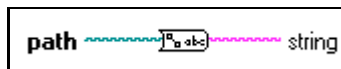
Path To Array Of Strings

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.



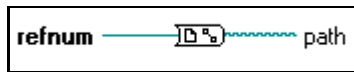
Path Type

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.



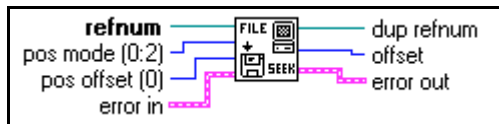
Refnum To Path

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



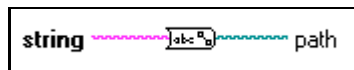
Seek

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**.



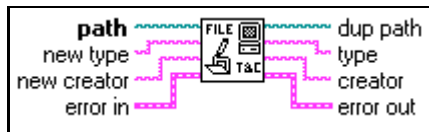
String To Path

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



Type and Creator

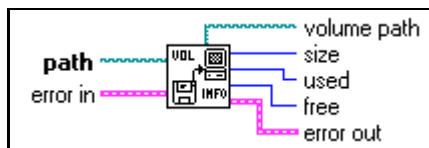
Reads and sets the type and creator of the file specified by **path**. File type and creator are four-character strings. If you do not specify **new type** or **new creator**, this function returns the current settings unchanged.



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 Type and Creator 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**.

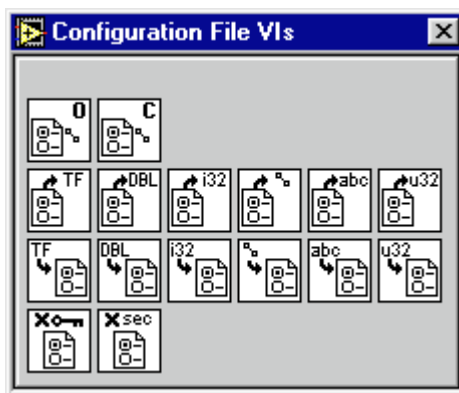
Volume Info

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.



Configuration File VIs

The Configuration File VIs provide you with the tools to create, modify, and read a platform-independent configuration file. The following illustration shows the options available on the **Configuration File VIs** subpalette.



The Configuration File VIs work with a platform-independent configuration file similar in format to the standard Windows initialization (.ini) file.

The file is divided into sections, denoted by a name enclosed in brackets. Each section in a file must have a unique name. Within each section are key and value pairs. Each key within a section must have a unique name.

An example of a configuration file with sections `section 1` and `section 2` is:

```
[section 1]
key1="string value 1"
key2="string value 2"
key3=53
[section 2]
key1=TRUE
key2=-12.3
key3="/c/temp/data.dat"
```

The Configuration File VIs support the following data types:

- Strings
- Paths
- Booleans
- 64-bit floating-point numbers (Double)
- 32-bit signed integers (I32)
- 32-bit unsigned integers (U32)

String data in the file must be enclosed in double quotes. Any unprintable characters in the string are stored in the file with their equivalent hexadecimal escape codes (for example, `\0D` for carriage return). In addition, backslash characters are stored in the file as double-backslashes (for example, `\\` for `\`).

Path data is stored in a platform-neutral format. This format is the standard UNIX format for paths. The VIs will interpret the absolute path `/c/temp/data.dat` as follows on the various G platforms:

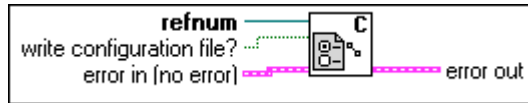
- Windows: `c\temp\data.dat`
- MacOS: `c:temp:data.dat`
- UNIX: `/c/temp/data.dat`

In addition, the VIs interpret the relative path `temp/data.dat` as follows:

- Windows: `temp\data.dat`
- MacOS: `:temp:data.dat`
- UNIX: `temp/data.dat`

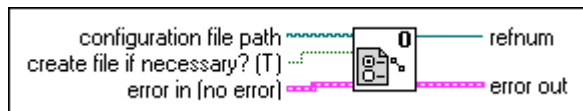
Close Config Data

Closes a reference to the configuration data identified by **refnum**. If **write configuration file?** is TRUE, the VI writes the data to the platform-independent configuration file identified by **refnum**.



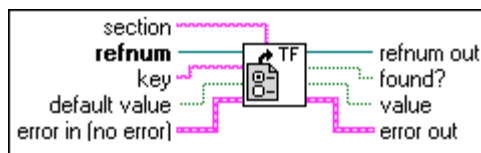
Open Config Data

Opens a reference to the configuration data found in a platform-independent configuration file. If the specified file does not exist and **create file if necessary?** is TRUE, the VI also creates the configuration file.



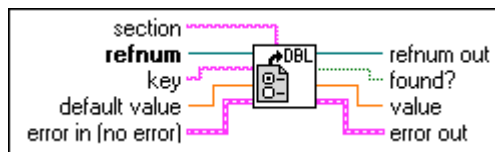
Read Key (Boolean)

Reads a Boolean **value** associated with a **key** in a specified **section** from the configuration data identified by **refnum**. If the **key** does not exist, the VI returns the **default value**.



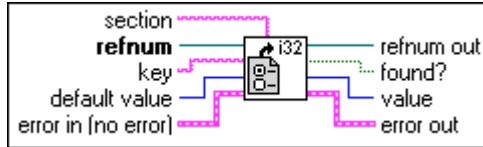
Read Key (Double)

Reads a 64-bit floating-point number **value** associated with **key** in a specified **section** from the configuration data identified by **refnum**. If **key** does not exist, the VI returns **default value**.



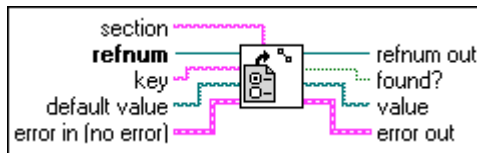
Read Key (I32)

Reads a 32-bit signed integer **value** associated with a **key** in a specified **section** from the configuration data identified by **refnum**. If the **key** does not exist, the VI returns the **default value**.



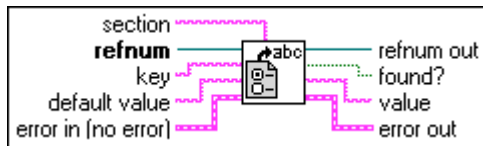
Read Key (Path)

Reads a path **value** associated with **key** in a specified **section** from the configuration data identified by **refnum**. If **key** does not exist, the VI returns **default value**.



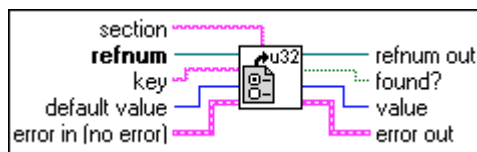
Read Key (String)

Reads a string **value** associated with **key** in a specified **section** from the configuration data identified by **refnum**. If **key** does not exist, the VI returns **default value**.



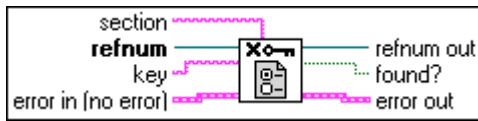
Read Key (U32)

Reads a 32-bit unsigned integer **value** associated with **key** in a specified **section** from the configuration data identified by **refnum**. If **key** does not exist, the VI returns the **default value**.



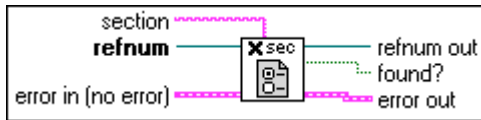
Remove Key

Removes a **key** in a specified **section** from the configuration data identified by **refnum**.



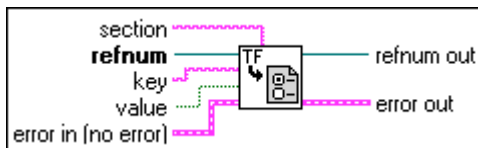
Remove Section

Removes a **section** from the configuration data identified by **refnum**.



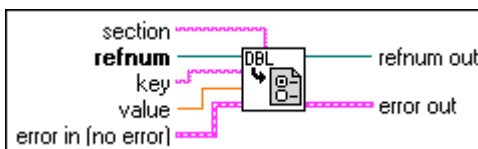
Write Key (Boolean)

Writes a Boolean **value** associated with **key** in a specified **section** of the configuration data identified by **refnum**. If **key** exists, the VI replaces the existing value. If **key** does not exist, the VI adds the **key/value** pair to the end of the specified **section**. If **section** does not exist, the VI adds **section**, with the **key/value** pair, to the end of the configuration data.



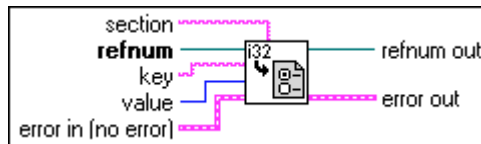
Write Key (Double)

Writes a 64-bit floating-point number **value** associated with **key** in a specified **section** of the configuration data identified by **refnum**. If **key** exists, the VI replaces the existing value. If **key** does not exist, the VI adds the **key/value** pair to the end of the specified section. If **section** does not exist, the VI adds the section, with the **key/value** pair, to the end of the configuration data.



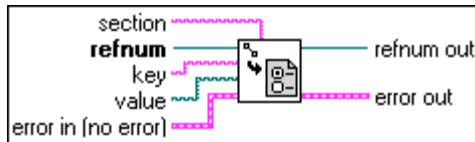
Write Key (I32)

Writes a 32-bit signed integer **value** associated with **key** in a specified **section** of the configuration data identified by **refnum**. If **key** exists, the VI replaces the existing value. If **key** does not exist, the VI adds the **key/value** pair to the end of the specified **section**. If **section** does not exist, the VI adds **section**, with the **key/value** pair, to the end of the configuration data.



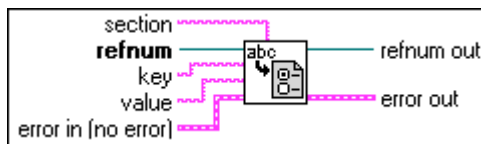
Write Key (Path)

Writes a path **value** associated with **key** in a specified **section** of the configuration data identified by **refnum**. If **key** exists, the VI replaces the existing value. If **key** does not exist, the VI adds the **key/value** pair to the end of the specified **section**. If **section** does not exist, the VI adds **section**, with the **key/value** pair, to the end of the configuration data.



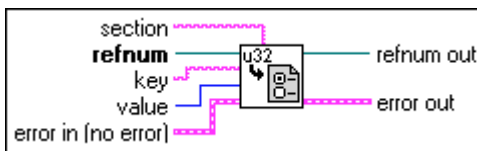
Write Key (String)

Writes a string **value** associated with **key** in a specified **section** of the configuration data identified by **refnum**. If **key** exists, the VI replaces the existing value. If **key** does not exist, the VI adds the **key/value** pair to the end of the specified **section**. If **section** does not exist, the VI adds **section**, with the **key/value** pair, to the end of the configuration data.



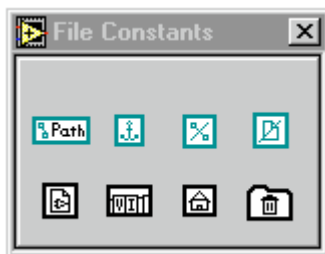
Write Key (U32)

Writes a 32-bit unsigned integer **value** associated with **key** in a specified **section** of the configuration data identified by **refnum**. If **key** exists, the VI replaces the existing value. If **key** does not exist, the VI adds the **key/value** pair to the end of the specified **section**. If the section does not exist, the VI adds **section**, with the **key/value** pair, to the end of the configuration data.



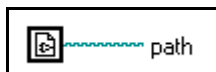
File Constants Descriptions

The following constants are available from 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 Preferences dialog box (**Edit>Preferences**), 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. You can set the value of the path constant to Not a Path by clicking on the path symbol with the Operating tool and selecting **Not a Path** from the resulting menu. See the *Paths and Refnums* section of Chapter 6, *Strings and File I/O*, in the *LabVIEW User Manual* for more information on using the Not a Path value.



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 G Preferences dialog box (**Edit»Preferences**), under **Paths**, defines this directory.



VI Library Constant

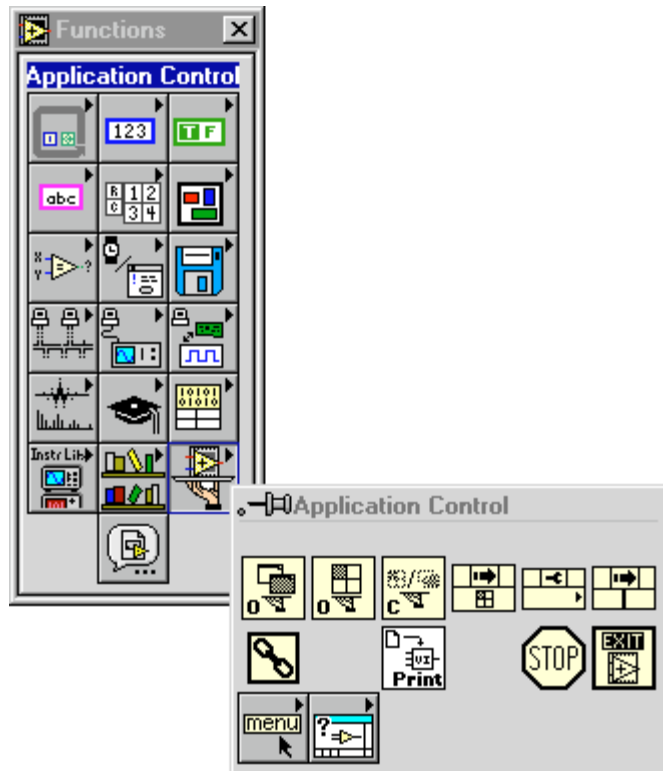
Returns the path to the VI library directory for the current development library on the current computer. The Preferences dialog box (**Edit»Preferences**), under **Paths**, defines this directory. If you build an application using the Application Builder libraries, this path is the path of the directory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



Application Control Functions

This chapter describes the Application Control functions.

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



The **Application Control** palette include the following subpalettes:

- Help functions
- Menu functions

Application Control Functions

The following Application Control functions are available.

Call By Reference Node

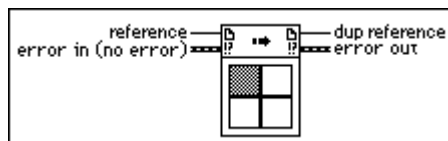
The Call By Reference node is very similar to a subVI node: you can use either to call a VI. However, there is a significant difference. With a subVI node, you determine what VI is called when you drop the node on the diagram.

With the Call By Reference node, the end user determines what VI is called at runtime via the **reference** input. The Call By Reference node could call a VI that resides on a different computer.

At the top of the Call By Reference node are four terminals: an input/output pair of flow through VI reference terminals, and an input/output pair of flow through error clusters. The VI reference input accepts wires only from strictly-typed VI references. Below these terminals is an area within which a connector pane resides that is identical to that of a VI with its terminals showing (rather than its icon). The connector pane of the strictly-typed VI reference input determines the pattern and data types of this connector pane. You should wire to these terminals just as you would to a normal subVI.

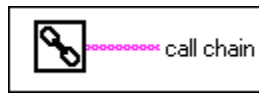
As long as none of the terminals of the connector pane have wires attached to them, the connector pane adapts automatically to the connector pane of the input VI reference. However, if any of them are wired, the node does not adapt automatically, and you must explicitly change the connector pane (possibly breaking those wires) by popping up on the node and selecting the **Adapt To Reference Input** menu item.

At run time there is a small amount of overhead in calling the VI that is not necessary in a normal subVI call. This overhead comes from validating the VI reference and a few other details. However, for a call to a VI in the local LabVIEW, this overhead should be insignificant for all but the smallest subVIs. Calling a VI located in another LabVIEW application (across the network) involves considerably more overhead. The **reference** input determines the VI that is called by the Call by Reference node.



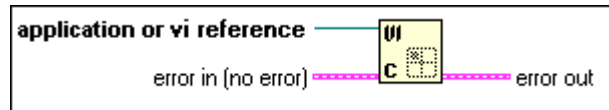
Call Chain

Returns a reference to a LabVIEW application or a VI.



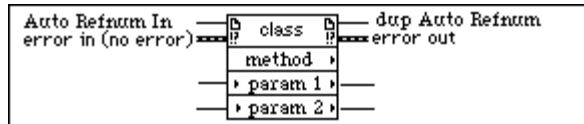
Close Application or VI Reference

Closes an open VI or the TCP connection to an open copy of LabVIEW.



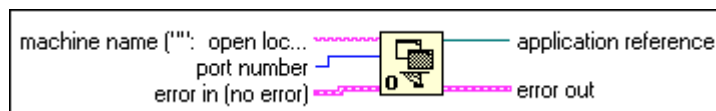
Invoke Node

Invokes a method or action on a VI. Most methods have parameters associated with them. To select the method, pop up anywhere on the node and select **Methods**. Once you select the method, the associated parameters appear in the following illustration. You can set and get the parameter values. Parameters with a white background are required inputs and the parameters with a gray background are recommended inputs.



Open Application Reference

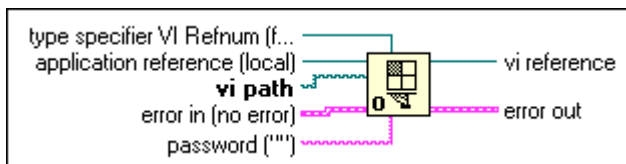
Returns a reference to a VI Server application running on the specified computer. If you do not specify a value for **machine name**, then it returns a reference to the local LabVIEW application in which this function is running.



You can use the **application reference** output as an input to the Property and Invoke nodes to get or set properties and invoke methods on the application. Using it as the input to the Open VI Reference function lets you get references to VIs in that application. Close the reference with the Close Application or VI Reference function. If you forget to close this reference, it closes automatically when the top-level VI associated with this function finishes executing. However, it is good practice to conserve the resources involved in maintaining the connection by closing the reference when you finish using it.

Open VI Reference

Returns a reference to a VI specified by a name string or path to the VI's location on disk.



You can get references to VIs in another LabVIEW application by wiring an **application reference** (obtained from the Open Application Reference function) to this function. In this case, **path input** refers to the file system on the remote LabVIEW computer. If you wire a reference to the local LabVIEW application you get the same behavior as if you had not wired anything to the **application reference** input.

If you intend to perform editing operations on the referenced VI, and the VI has a password-protected diagram, you can provide the password to the **password** string input. If you provide the incorrect password, the Open VI Reference function returns an error and an invalid VI reference. If you provide no password when opening a reference to a VI that is password protected, you can still get the reference, but you can only perform operations that do not edit the VI.

If you intend to call the specified VI through the Call By Reference function, wire a strictly-typed VI reference to the **type specifier** input. The function ignores the value of this input. Only the input's type—the connector pane information—is used. By specifying this type, the Open VI Reference function verifies at run time that the referenced VI's connector pane matches that of the **type specifier** input.



Note *It is possible to wire a Generic VI refnum type to the type specifier input. Doing this results in the same behavior as if you had not wired the type specifier input at all.*

If you wire the type specifier input with a strictly-typed VI refnum, the VI must meet several requirements before the VI reference is returned successfully:

- The VI cannot be broken for any reason.
- The VI must be runnable as a subVI; that is, it cannot be active as a top-level VI (unless the VI is re-entrant).
- The connector pane of the VI must match that of the type specifier.

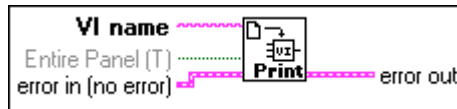
If you forget to close this reference, it closes automatically when the top-level VI associated with this function finishes executing. However, it is good practice to conserve the resources involved in maintaining the connection by closing the reference when you finish using it.

If you get a strictly-typed reference to a reentrant VI, a dedicated data space is allocated for that reference. This data space is always used in conjunction with the output VI reference. This can lead to some new behaviors that you may not be accustomed to in LabVIEW. For example, parallel calls (using the Call By Reference node) to a reentrant VI using the same VI reference do not execute in parallel, but execute serially, one after the other.

Notice that a VI reference is similar to what is known as a function pointer in other languages. However, in LabVIEW, these function pointers also can be used to call VIs across the network.

Print Panel

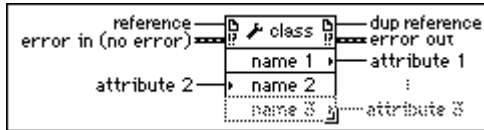
Produces the same printout as programmatic print at completion, but can be called from other VIs and at times other than at completion. By default, it prints the entire panel, not just what is visible in the window. This VI assumes that the VI is loaded but does not require the window to be open.



Property Node

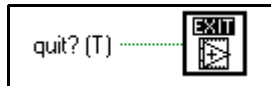
Sets (writes) or gets (reads) VI and application property information. To select the VI or application class, pop up on the node and select from the **Select VI Server Class** submenu. To select an application class, select **Application**. To select a VI class, select **Virtual Instrument**, or wire the VI or application refnum to **reference** and the node choices change accordingly.

To select a specific property, pop up on one of the **name** terminals and select **Properties**. To set property information, pop up and select **Change to Write**, and to get property information pop up and select **Change to Read**. Some properties are read only, so you cannot see **Change to Write** in the pop-up menu. The Property node works the same way as Attribute nodes. If you want to add items to the node, pop up and select **Add Element** or click and drag the node to expand the number of items in the node. When this node executes, properties are handled in the order from top to bottom. If an error occurs on one of the properties, the node stops at that property and returns an error. No further properties are handled. The error string reports which property caused the error. Remember if the small direction arrow on a property is on the left, you are setting the property value. If the small direction arrow on the property is on the right, you are getting the property value. Each property name has a short or long name which can be changed by popping up and selecting **Name Format**. Another name format is no name where only the type is displayed for each property.



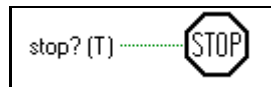
Quit

Stops all executing VIs and ends the current session of LabVIEW. This function shuts down only LabVIEW; the function does not affect other applications. The function stops all 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, avoid using the Stop function when you have a built-in termination 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.

Help Function Descriptions

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



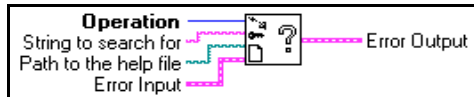
Control Help Window

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.



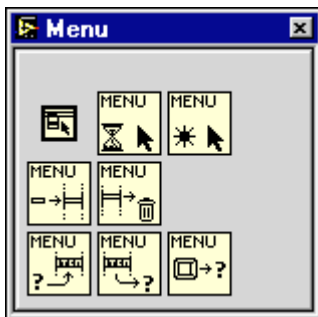
Get Help Window Status

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



Menu Functions

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

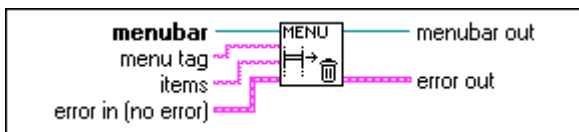


The Menu functions operate on menus identified by a refnum. A VI's menu refnum is obtained through the constant Current VI's menu. Items are identified by an item tag (string) and sometimes by an item path (string), which is a list of item tags from the menu tree root up to the item and separated by colons.

The following Menu functions are available.

Delete Menu Items

Deletes menu items from the menubar or a submenu within the menubar.



If **menu tag** is specified, the items are deleted from the submenu specified by **menu tag**, or else the items are deleted from the menubar. The function returns an error if **menu tag** or one of the items specified is not found.

items can be a tag (string) of an existing item, an array of tags of existing items, a position index (zero-based integer) of an item in the menu or an array of position indices of items in the menu. If you do not wire **items**, all the items in the menu are deleted. If there is a submenu in any of the specified items, the submenu and all its contents are deleted automatically. Because separators do not have unique tags, they are best deleted by using their positional indices.

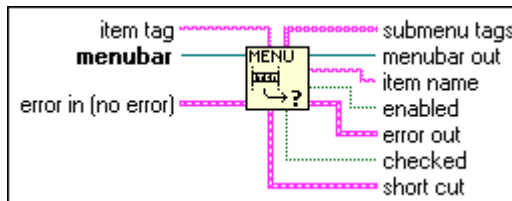
Enable Menu Tracking

Enables or disables tracking of menu selections.



Get Menu Item Info

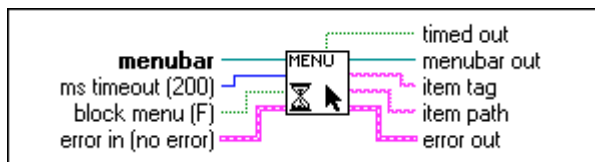
Returns the attributes of the menu item specified through **item tag**.



Item attributes are **item name** (the string that appears in the menu), **enabled** (false designates that the item is grayed out), **checked** (specifies whether there is a check mark next to the item), and **short cut** (key accelerator). If the item has a submenu, its item tags are returned as an array of strings in **submenu tags**. If **item tag** is unwired, the menubar items are returned. If **item tag** is not valid, an error is returned.

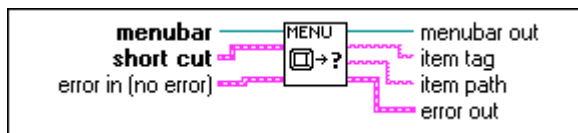
Get Menu Selection

Returns the **item tag** of the last selected menu item, optionally waiting **timeout** milliseconds. **item path** is a string describing the position of the item in the menu hierarchy, which is the format of a list of menu tags separated by a colon (:). If **block menu** is set to **True**, Menu selection is blocked out after an item tag is read.



Get Menu Shortcut Info

Returns the menu item that is accessible through a given shortcut.

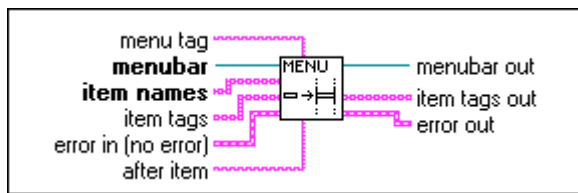


item path is a string of menu item tags separated by a colon (:).

short cut consists of a string (key) and a Boolean (specifies whether the shift key is included or not).

Insert Menu Items

Inserts menu items into a menubar or a submenu within the menubar.



menu tag specifies the submenu where items are inserted. If you do not specify **menu tag**, the items are inserted into the **menubar**.

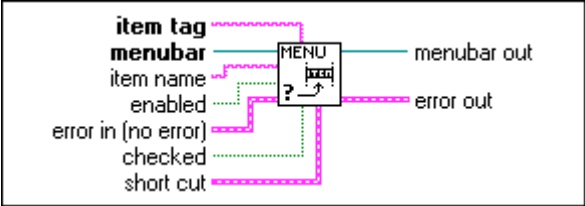
item names and **item tags** identify the items to be inserted into the menu. The type of **item names** and **item tags** can be an array of strings (for inserting multiple items) or just a string (for inserting a single item). You can wire in either **item names** or **item tags**, in which case both names and tags get the same values. If you require each item to have different name and tag, you must wire in separate values for **item names** and **item tags**.

after item specifies the position where the items are inserted. **after item** can be a tag (string) of an existing item or a position index (zero based integer) in the menu. To insert at the beginning of the menu, wire a number less than 0 to **after item**. To insert at the end of the menu, wire a number larger than the number of items in the menu. You can insert a separator using the application tag APP_SEPARATOR. The function always ensures that the tags of all the inserted menu items are unique to the menu hierarchy by appending numbers to the supplied tags, if necessary.

item tags out returns the actual tags of the inserted items. If **menu tag** or **after item** (tag) is not found, the function returns an error.

Set Menu Item Info

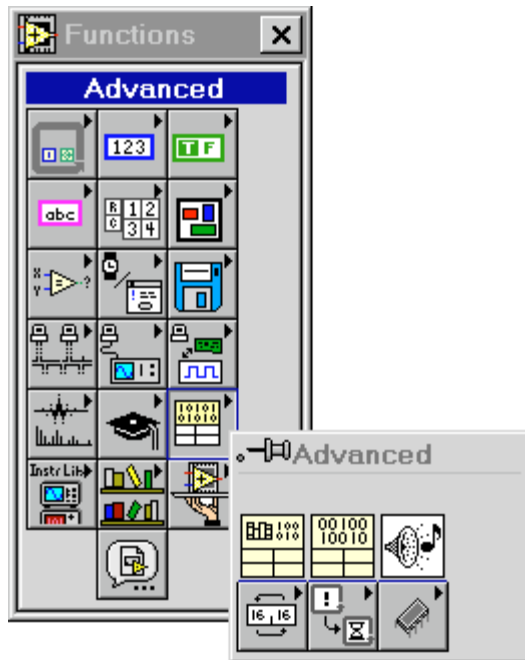
Sets the attributes of a menu item specified through **menu** and **item tag**. Item attributes are **item name** (the string that appears in the menu), **enabled** (false designates that the item is grayed out), **checked** (specifies whether there is a check mark next to the item), and **shortcut** (key accelerator). Attributes that are not wired remain unchanged. If **item tag** is not valid, an error is returned.



Advanced Functions

This chapter describes the functions that perform advanced operations. This chapter also describes the Data Manipulation and Synchronization functions, and the VI Control and Memory VIs.

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



The Advanced functions include the following subpalettes:

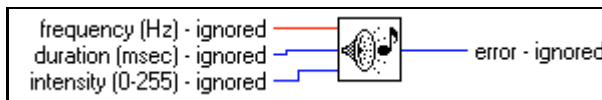
- Data Manipulation
- Memory
- Synchronization
- VI Control

Advanced Function Descriptions

The following Advanced functions are available.

Beep

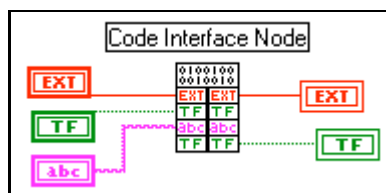
Causes the system to issue an audible tone. You can specify the tone frequency in Hertz, the duration in milliseconds, and the intensity as a value from 0 to 255, with 255 being the loudest. Although this VI appears on all platforms, the frequency, duration, and intensity parameters work only on the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



Code Interface Node

Calls code written in a conventional programming language, such as C, directly from a block diagram. Code Interface Nodes (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.

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



The LabVIEW 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`.

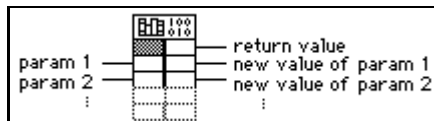
For more information on the Code Interface Node, see the *LabVIEW Code Interface Reference Manual*, available in portable document format (PDF) only.

Call Library Function

Calls 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 can 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.

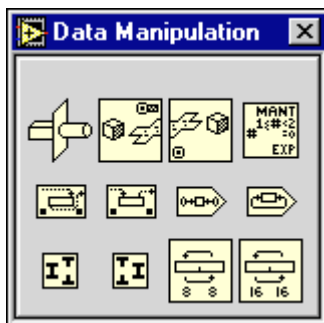


The Call Library Function consists of paired input/output terminals with input on the left and output on the right. You can use one or both. The return value for the function is returned in the right terminal of the top pair of terminals of the node. If there is no return value, then this pair of terminals 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 25, *Calling Code From Other Languages*, in the *G Programming Reference Manual*.

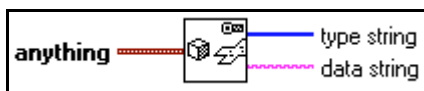
Data Manipulation Function Descriptions

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



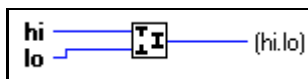
Flatten To String

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 *G Programming Reference Manual*.



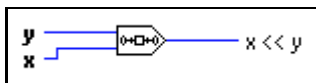
Join Numbers

Creates a number from the component bytes or words.



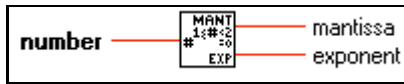
Logical Shift

Shifts x the number of bits specified by y .



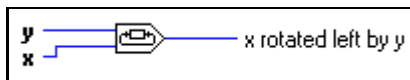
Mantissa & Exponent

Returns the mantissa and exponent of the input numeric value such that **number** = **mantissa** * 2^{**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.



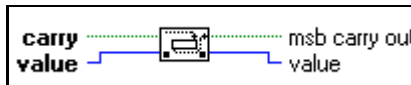
Rotate

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



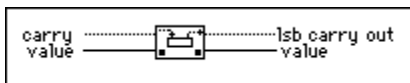
Rotate Left With Carry

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.



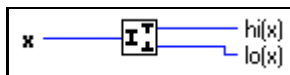
Rotate Right With Carry

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.

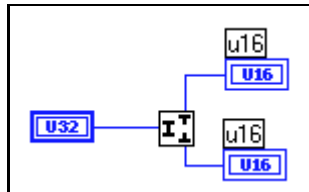
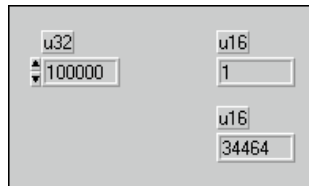


Split Number

Breaks a number into its component bytes or words.

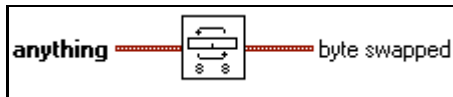


The following illustration shows an example of how to use the Split Number function. The function splits the signed 32-bit number 100,000 into the high word component, 1, and the low word component, 34,464.



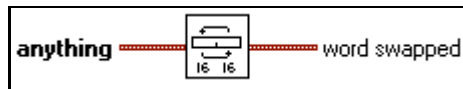
Swap Bytes

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



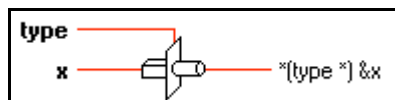
Swap Words

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**.

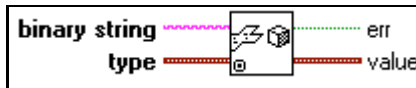


Casting data to a string converts it into machine-independent, big endian form. That is, the function puts the most significant byte or word first and the least significant byte or word last, removes alignment, and converts extended-precision numbers to 16 bytes. Casting a string to

a 1D array converts the string from machine-independent form to the native form for that platform.

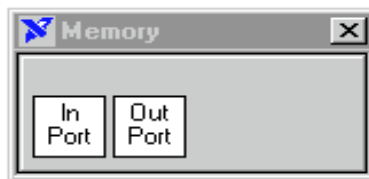
Unflatten From String

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 *G Programming Reference Manual*.



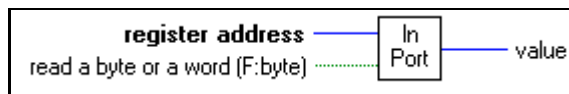
Memory VI Descriptions

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



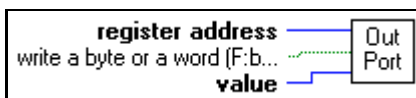
In Port (Windows 3.1 and Windows 95)

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)

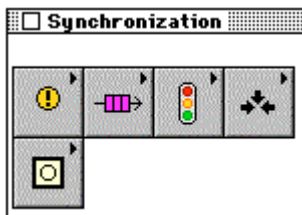
Writes a byte or word integer to a specific **register address**. Because this VI is not available on all platforms, VIs using this subVI are not portable.



Synchronization VIs

You can synchronize tasks executing in parallel by using the Synchronization VIs. You can also use the Synchronization VIs to pass data between parallel tasks. You access the **Synchronization** palette by choosing **Functions»Advanced»Synchronization**.

The following illustration displays the options available on the **Synchronization** palette.



The **Synchronization** palette consists of five subpalettes:

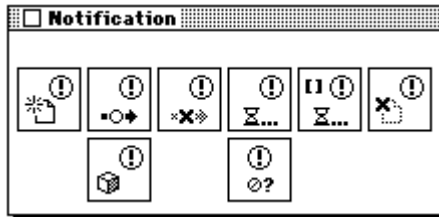
- Notification VIs
- Queue VIs
- Rendezvous VIs
- Semaphore VIs
- Occurrence Functions

Notification VIs

You can use the Notification VIs to pass data from one task to one or more separate, parallel tasks. In particular, you use these VIs when you want one or more VIs or parts of block diagrams to wait until another VI or part of a block diagram sends them some data.

The Notification VIs differ from the Queue VIs in that the data sent is not buffered. That is, if there is no one waiting on a notification when it is sent, the data will be “lost” if another notification is sent. Also, more than one Wait On Notification VI can receive the same data.

You can access the notification VIs by selecting **Functions»Advanced»Synchronization»Notification**.



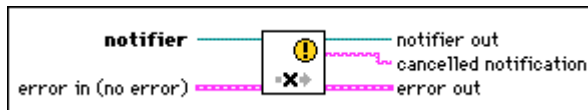
The notification VIs use the **Notifier RefNum** control from the **Controls»Path & Refnum** palette.



The Notifier RefNum can be used with the following VIs.

Cancel Notification

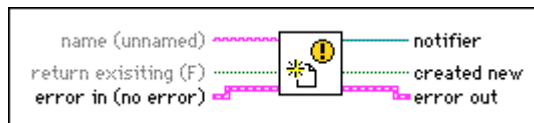
Cancels and returns a previously sent notification.



This prevents a call to the Wait On Notification VI with **ignore previous** set to FALSE to see the previously sent notification.

Create Notifier

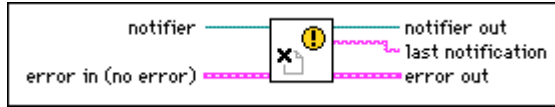
Looks up an existing **notifier** or creates a new **notifier** and returns a refnum that you can use when calling other Notification VIs.



If **name** is specified, the VI first searches for an existing **notifier** with the same **name** and will return its refnum if it exists. If a named notifier with the same name does not already exist and the **return existing** input is FALSE, the VI will create a new **notifier** and return its refnum. The **created new** output returns TRUE if the VI creates a new notifier.

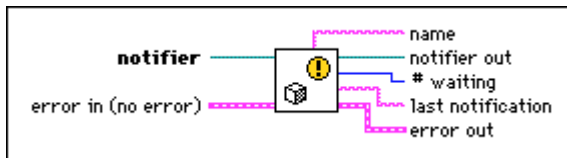
Destroy Notifier

Destroys the specified **notifier** and returns the **last notification** that was sent. All Wait on Notification VIs that are currently waiting on this notifier time out immediately and return an error.



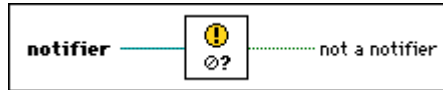
Get Notifier Status

Returns current status information of **notifier**.



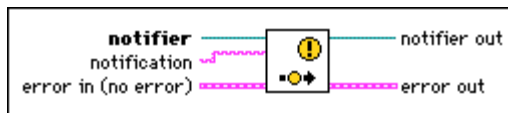
Not A Notifier

Returns TRUE if **notifier** is not a valid notifier refnum.



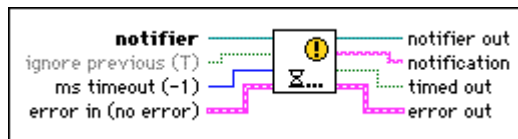
Send Notification

Sends **notification** to the specified **notifier**. All Wait On Notification VIs that are currently waiting on this **notifier** stop waiting and return the specified **notification**.



Wait On Notification

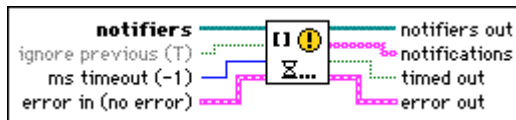
Waits for the Send Notification VI to send **notification** to the specified notifier.



If **ignore previous** is FALSE and a notification was sent since the last time this VI was called, the VI returns immediately with the value of the old notification and with **timed out** as FALSE. If the **ignore previous** input is TRUE, the VI will wait **timeout** milliseconds (default -1, or forever) before timing out. If a notification is sent, **timed out** will return FALSE. If a notification is not sent or if **notifier** is not valid, **timed out** will return TRUE.

Wait On Notification From Multiple

Waits for the Send Notification VI to send a notification to one of the specified notifiers.



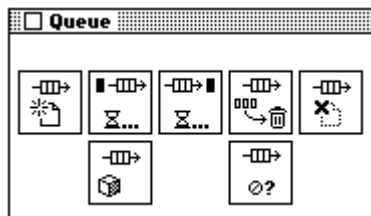
If **ignore previous** is FALSE and a notification was sent to any of the specified notifiers since the last time this VI was called, the VI returns immediately with the value(s) of the old notification(s) and with **timed out**=FALSE. If the **ignore previous** input is TRUE, the VI will wait **ms timeout** milliseconds (default -1, or forever) before timing out. If at least one notification is sent, **timed out** will return FALSE. If no notification is sent, **timed out** will return TRUE.

Queue VIs

You can use the Queue VIs to pass an ordered sequence of data elements from one task to another separate, parallel task. In particular, you use these VIs when you want one task to wait until another task provides it with some data. You can also use these VIs when you want one task to wait until another task has processed some data that the first task has provided.

The queue VIs differ from the notification VIs in that the data sent is buffered. That is, if there is no one waiting to read from the queue when an element is inserted, the element stays in the queue until it is explicitly removed. Also, when data is inserted into a queue and there are two VIs waiting to remove it from the queue, only one of them receives the data.

You can access the Queue VIs by selecting **Functions»Advanced»Synchronization»Queue**.



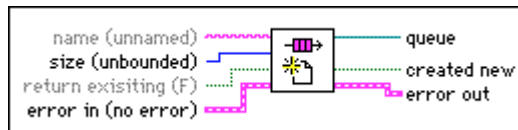
The Queue VIs use the **Queue RefNum** control from the **Controls»Path & Refnum** palette.



Queue RefNum can be used with the following VIs.

Create Queue

Looks up an existing queue or creates a new queue and returns a refnum that you can use when calling other queue VIs.

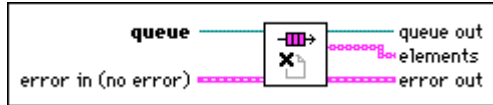


If you specify a size > 0 , the queue size is limited to that many elements. If the Insert Queue Element VI tries to insert an element into a full queue, it must wait until an element is removed with the Remove Queue Element VI. The default size is -1 for an unbounded queue.

If a name is specified, the VI first searches for an existing queue with the same name and will return its refnum if it exists. If a named queue with the same name does not already exist and the **return existing** input is FALSE, the VI creates a new queue and return its refnum. The **created new** output returns TRUE if the VI creates a new queue.

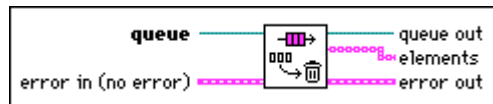
Destroy Queue

Destroys the specified queue and returns any elements that are in the queue. All Insert Queue Element and Remove Queue Element VIs that are currently waiting on this queue time out immediately and return an error.



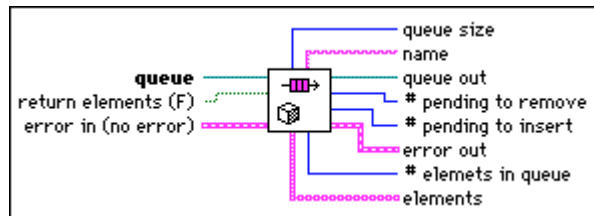
Flush Queue

Removes all **elements** from **queue**.



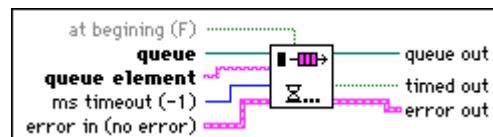
Get Queue Status

Returns current status information of **queue**.



Insert Queue Element

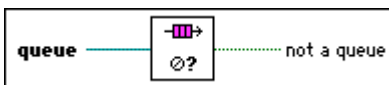
Inserts an element into a queue.



The **at beginning** parameter specifies whether the element is inserted at the end (default) or the front of the queue. If the queue is full, the VI waits **timeout** milliseconds (default –1, or forever) before timing out. If space becomes available during the wait, the element is inserted and **timeout** returns FALSE. If the queue remains full or the queue is not valid, **timeout** returns TRUE.

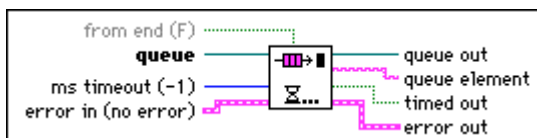
Not A Queue

Returns TRUE if **queue** is not a valid queue refnum.



Remove Queue Element

Removes an element from a queue.

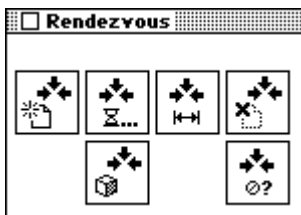


The **from end** parameter specifies whether the returned element is taken from the front (default) or the end of the queue. If the queue is empty, the VI waits **timeout** milliseconds (default -1 , or forever) before timing out. If an element becomes available during the wait, the element is returned and **timed out** returns FALSE. If no element becomes available or the queue is not valid, **timed out** returns TRUE.

Rendezvous VIs

You can use the Rendezvous VIs to synchronize two or more separate, parallel tasks at specific points of execution. Each task that reaches the rendezvous waits until the specified number of tasks are waiting, at which point all tasks proceed with execution.

You can access the Rendezvous VIs by selecting **Functions»Advanced»Synchronization»Rendezvous**.



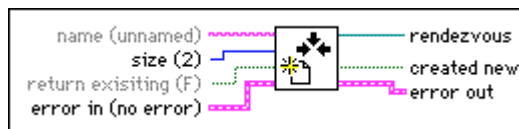
The Rendezvous VIs use the **Rendezvous RefNum** control from the **Controls» Path & Refnum** palette.



The Rendezvous RefNum can be used with the following VIs.

Create Rendezvous

Looks up an existing **rendezvous** or creates a new **rendezvous** and returns a refnum that you can use when calling other Rendezvous VIs.

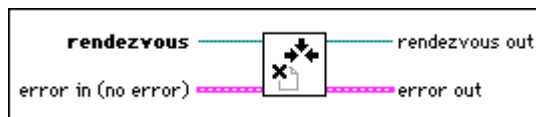


The **size** specifies how many tasks have to meet at the **rendezvous** in order to continue execution. The default size is 2.

If **name** is specified, the VI first searches for an existing **rendezvous** with the same name and returns its refnum if it exists. If a named rendezvous with the same name does not already exist and the **return existing** input is FALSE, the VI creates a new rendezvous and return its refnum. The **created new** output returns TRUE if the VI creates a new rendezvous.

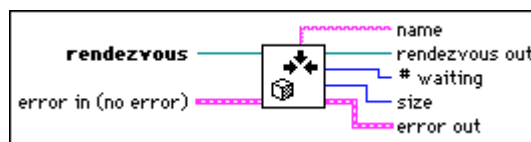
Destroy Rendezvous

Destroys the specified rendezvous. All Wait at Rendezvous VIs that are currently waiting on this rendezvous time out immediately and return an error.



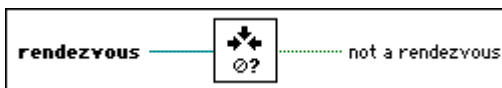
Get Rendezvous Status

Returns current status information of a **rendezvous**.



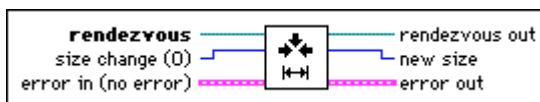
Not A Rendezvous

Returns TRUE if **rendezvous** is not a valid rendezvous refnum.



Resize Rendezvous

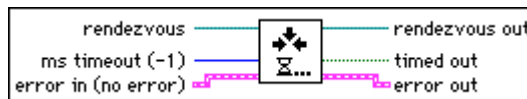
Changes the size of **rendezvous** by **size change** and returns **new size**.



If the number of tasks currently waiting at **rendezvous** is less than or equal to **new size**, the first size tasks stop waiting and continue execution.

Wait at Rendezvous

Waits until a sufficient number of tasks have arrived at the rendezvous.



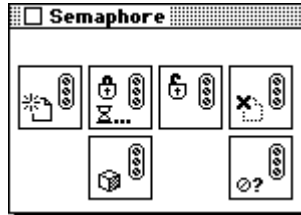
If the number of tasks, including the new one, waiting at **rendezvous** is less than the rendezvous size, the VI waits **timeout** milliseconds (default -1 , or forever) before timing out. If enough tasks arrive at the rendezvous during the wait, **timed out** returns FALSE. If enough tasks do not arrive or the rendezvous is not valid, **timed out** returns TRUE.

Semaphore VIs

Semaphores, also known as Mutex, are used to limit the number of tasks that may simultaneously operate on a shared (protected) resource. A protected resource or critical section may include writing to global variables or communicating with external instruments.

You can use the Semaphore VIs to synchronize two or more separate, parallel tasks so that only one task at a time executes a critical section of code protected by a common semaphore. In particular, you use these VIs when you want other VIs or parts of block diagram to wait until another VI or part of a block diagram is finished with the execution of a critical section.

You can access the Semaphore VIs by selecting **Functions»Advanced»Synchronization»Semaphore**.



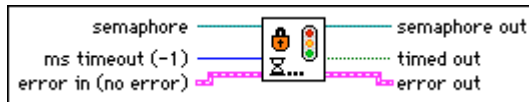
The semaphore VIs use the **Semaphore RefNum** control from the **Controls» Path & Refnum** palette.



The Semaphore RefNum can be used with the following VIs.

Acquire Semaphore

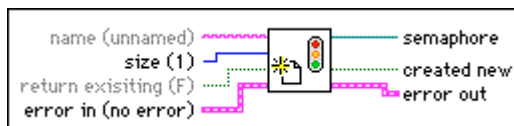
Acquires access to a semaphore.



If the semaphore is already acquired by the maximum number of tasks, the VI waits **timeout** milliseconds (default -1 , or forever) before timing out. If the semaphore becomes available during the wait, **timed out** returns FALSE. If the semaphore does not become available or the semaphore is not valid, **timed out** returns TRUE.

Create Semaphore

Looks up an existing semaphore or creates a new semaphore and returns a refnum that you can use when calling other semaphore VIs.



size specifies how many tasks may acquire the semaphore at the same time. The default size is 1.

If a name is specified, the VI first searches for an existing semaphore with the same name and returns its refnum if it exists. If a named semaphore with the same name does not already exist

and the **return existing** input is FALSE, the VI creates a new semaphore and return its refnum. The **created new** output returns TRUE if the VI creates a new semaphore.

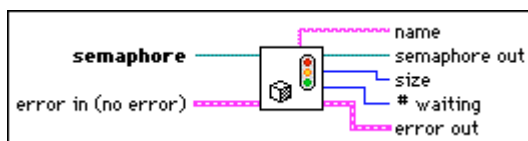
Destroy Semaphore

Destroys the specified semaphore. All Acquire Semaphore VIs that are currently waiting on this semaphore will time out immediately and return an error.



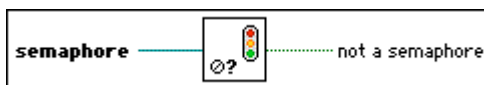
Get Semaphore Status

Returns current status information of a semaphore.



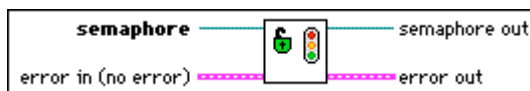
Not A Semaphore

Returns TRUE if **semaphore** is not a valid semaphore refnum.



Release Semaphore

Releases access to a semaphore.



If there is an Acquire Semaphore VI waiting for this semaphore, it stops waiting and continues execution. If you call the Release Semaphore VI on a semaphore that you have not acquired, you effectively increment the semaphore size.

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.

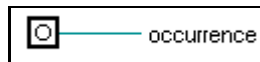
You can perform the same task using global variables, with one loop polling the value of the global until its value changes. However, global variables add overhead, because the loop that pulls the global variable uses execution time. With occurrences, the polling loop is replaced with a Wait on Occurrence 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.

Unlike other synchronization VIs, 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.

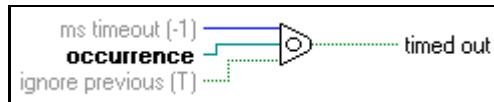
Set Occurrence

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**.



Data Acquisition VIs

Part II, *Data Acquisition VIs*, introduces the collection of VIs that work with your data acquisition (DAQ) hardware devices. This part contains the following chapters:

- Chapter 14, *Introduction to the LabVIEW Data Acquisition VIs*, contains basic information about the data acquisition (DAQ) VIs and shows where you can find them in LabVIEW.
- Chapter 15, *Easy Analog Input VIs*, describes the Easy Analog Input VIs, which perform simple analog input operations.
- Chapter 16, *Intermediate Analog Input VIs*, describes the Intermediate Analog Input VIs.
- Chapter 17, *Analog Input Utility VIs*, 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.
- Chapter 18, *Advanced Analog Input VIs*, 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.
- Chapter 19, *Easy Analog Output VIs*, describes the Easy Analog Output VIs in LabVIEW, which perform simple analog output operations.
- Chapter 20, *Intermediate Analog Output VIs*, 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.
- Chapter 21, *Analog Output Utility VIs*, 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.

- Chapter 22, *Advanced Analog Output VIs*, 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.
- Chapter 23, *Easy Digital I/O VIs*, describes the Easy Digital I/O VIs, which perform simple digital I/O operations.
- Chapter 24, *Intermediate Digital I/O VIs*, describes the Intermediate Digital I/O VIs. These VIs are single VI solutions to common digital problems.
- Chapter 25, *Advanced Digital I/O VIs*, 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.
- Chapter 26, *Easy Counter VIs*, describes the Easy Counter VIs that perform simple counting operations.
- Chapter 27, *Intermediate Counter VIs*, describes Intermediate Counter VIs you can use to program counters on MIO, TIO, and other devices with the DAQ-STC or Am9513 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 the Intermediate Counter VIs for several pulse generation, counting, and measurement operations.
- Chapter 28, *Advanced Counter VIs*, 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.
- Chapter 29, *Calibration and Configuration VIs*, describes the VIs that calibrate specific devices and set and return configuration information.
- Chapter 30, *Signal Conditioning VIs*, 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.

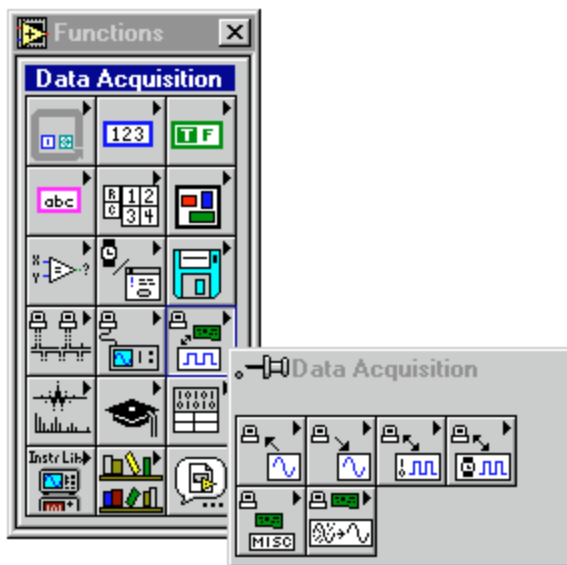
Introduction to the LabVIEW Data Acquisition VIs

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.

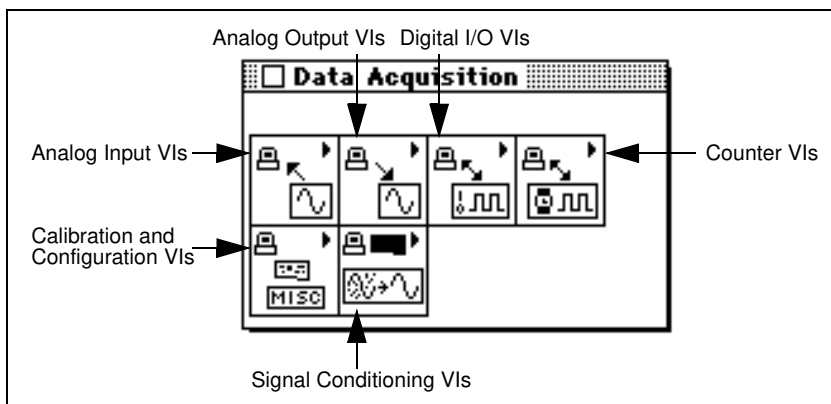
LabVIEW includes a collection of VIs that work with your DAQ hardware devices. With LabVIEW DAQ VIs you can develop acquisition and control applications.

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 part 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 VIs 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 5, *Printing and Documenting VIs*, of the *G Programming Reference Manual*.

**Note**

Use only the inputs 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.

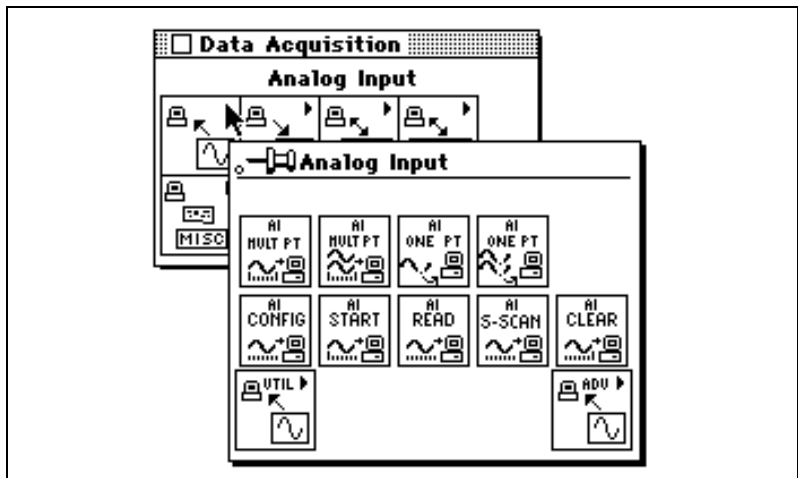
**Note**

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.

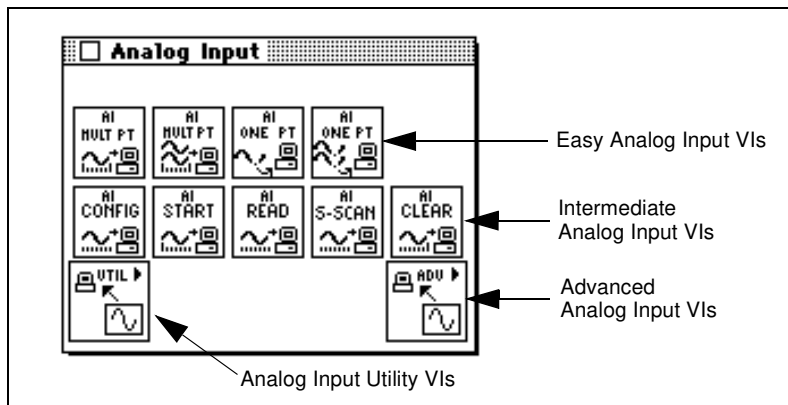
The Analog Input VIs

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

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 provide a basic, convenient interface with only the most commonly used inputs and outputs. For more complex applications, you should use the Intermediate Analog Input VIs and Advanced Analog Input VIs for more functionality and performance.

Refer to Chapter 15, *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 is 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 16, *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 17, *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 18, *Advanced Analog Input VIs*, for specific VI information.

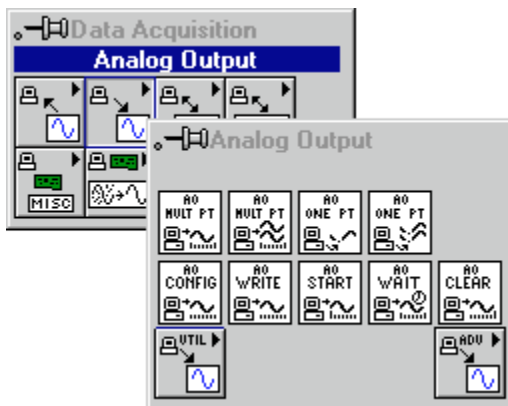
Locating Analog Input VI Examples

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

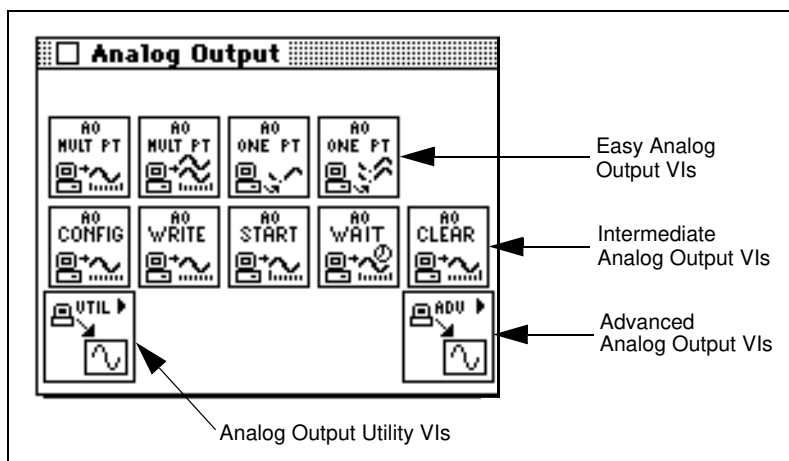
Analog Output VIs

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 provide a basic, convenient interface with only the most commonly used inputs and outputs. For more complex applications, you should use the Intermediate Analog Output VIs and Advanced Analog Output VIs for more functionality and performance.

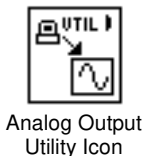
Refer to Chapter 19, *Easy Analog Output VIs*, for specific VI information.

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 is 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 20, *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 21, *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 22, *Advanced Analog Output VIs*, for additional information on the inputs and outputs and how they work.

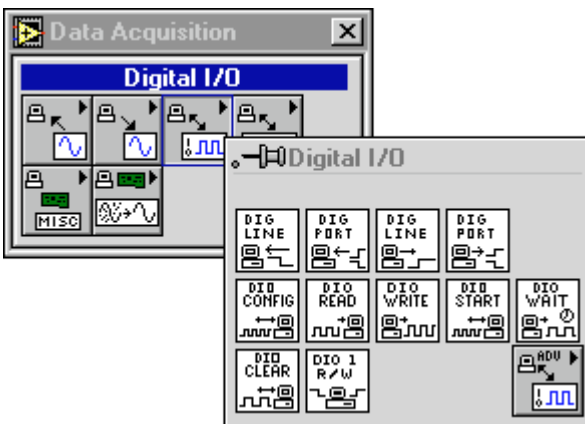
Locating Analog Output VI Examples

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

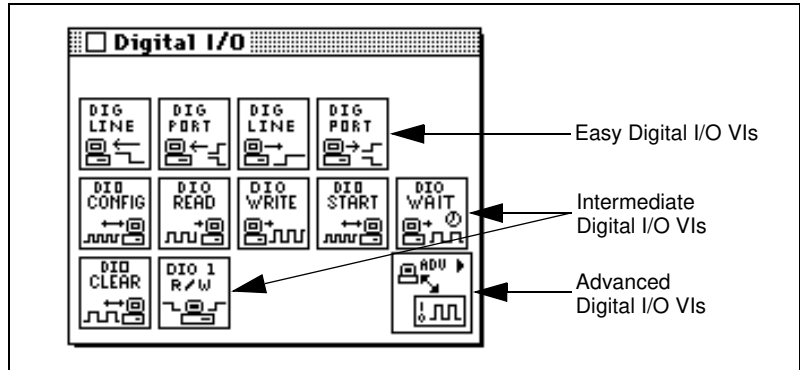
Digital Function VIs

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 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 23, *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 24, *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 25, *Advanced Digital I/O VIs*, for additional information on the inputs and outputs and how they work.

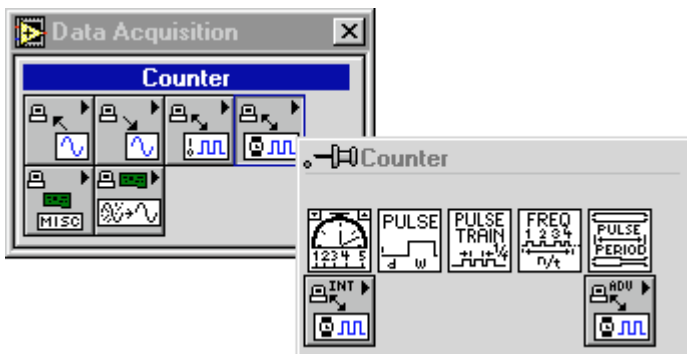
Locating Digital I/O VI Examples

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

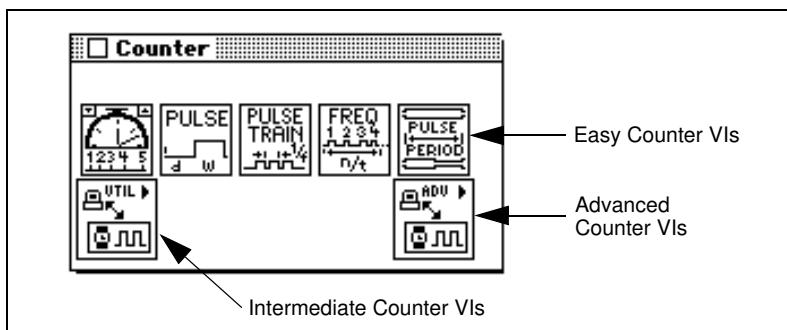
Counter VIs

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.

Refer to Chapter 26, *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 27, *Intermediate Counter VIs*, for specific VI information.

Advanced Counter VIs



Advanced Counter VI Icon

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 28, *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 libraries by opening `examples\daq\counter\DAQ-STC.llb`, `examples\daq\counter\am9513.llb`, and `examples\daq\counter\8253.llb`.

Calibration and Configuration VIs

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

See Chapter 29, *Calibration and Configuration VIs*, for information on locating these VIs and examples.

Signal Conditioning VIs

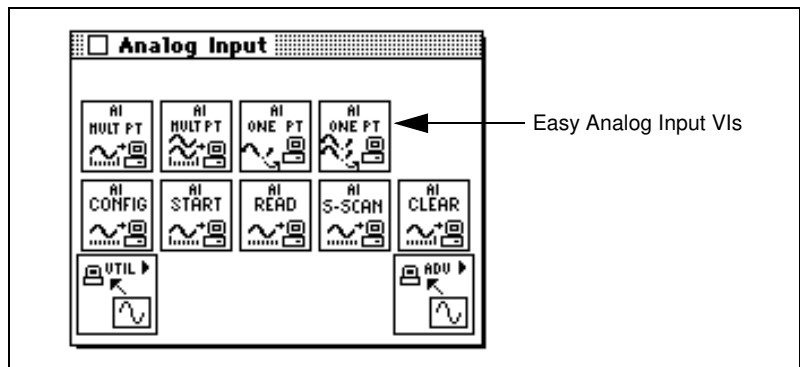
These VIs convert analog input voltages read from resistance temperature detectors (RTDs), strain gauges, or thermocouples into units of strain or temperature.

See Chapter 30, *Signal Conditioning VIs*, for information on locating these VIs and examples.

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.

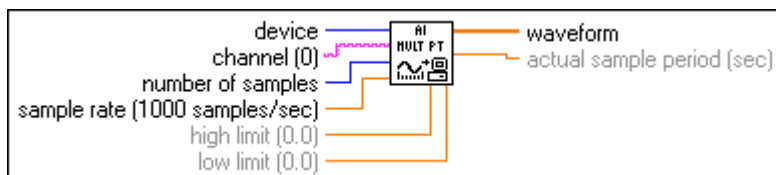


Easy Analog Input VI Descriptions

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.

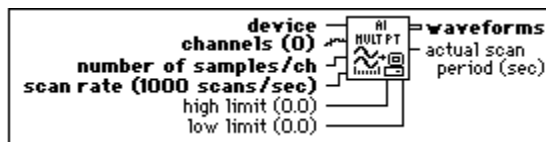


The AI Acquire Waveform VI performs a hardware-timed measurement of a waveform (multiple voltage readings at a specified sampling rate) on a single analog input channel. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

Refer to Appendix B, [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.

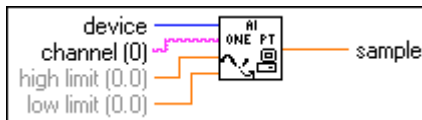


The AI Acquire Waveforms VI performs a timed measurement of multiple waveforms on the specified analog input channels. If an error occurs, a dialog box appears, giving you the option to abort the operation or continue execution.

Refer to Appendix B, [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 value.

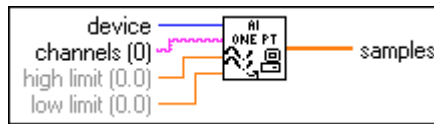


The AI Sample Channel VI performs a single, untimed measurement of a channel. If an error occurs, a dialog box appears giving you the option to stop the VI or continue.

Refer to Appendix B, [DAQ Hardware Capabilities](#), for the channel numbers and input limits available with your DAQ device.

AI Sample Channels

Performs a single reading from each of the specified channels.



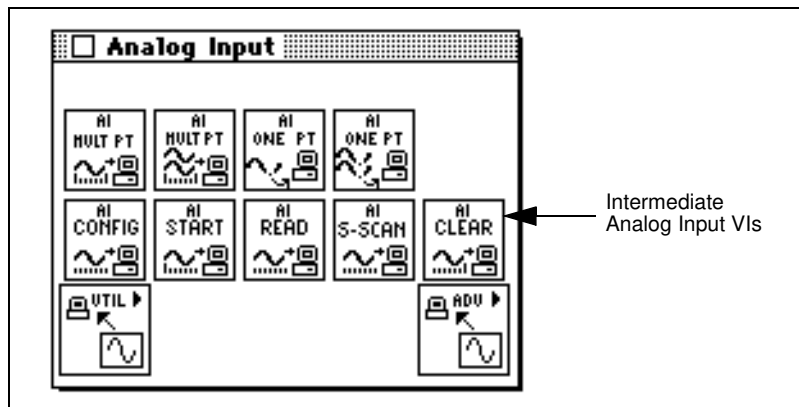
The AI Sample Channels VI measures a single value from each of the specified analog input channels. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

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

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.



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.

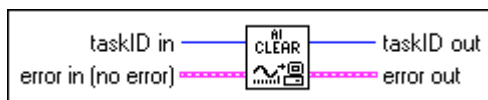
The General Error Handler VI is in **Functions»Time and Dialog** in LabVIEW.

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 18, *Advanced Analog Input VIs*, for description of the AI Control VI.



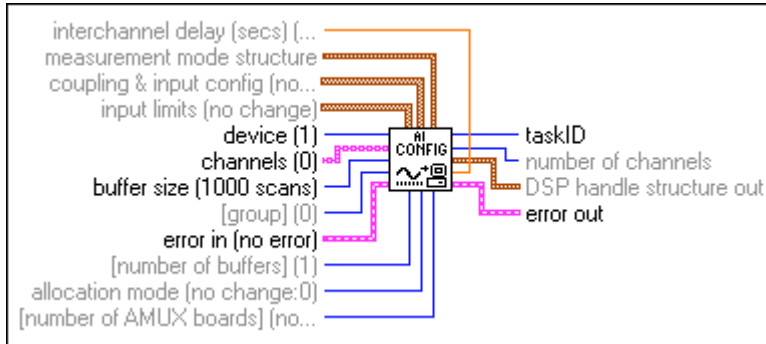
Note *The AI Clear VI always clears the acquisition regardless of whether error in indicates that an error occurred.*

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 Chapter 10, *Time, Dialog, and Error Functions*.

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.



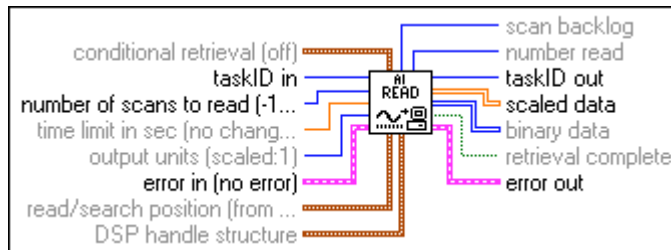
You can allocate more than one buffer only with the following devices.

- **(Macintosh)** NB-A2000, NB-A2100, and NB-A2150

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

AI Read

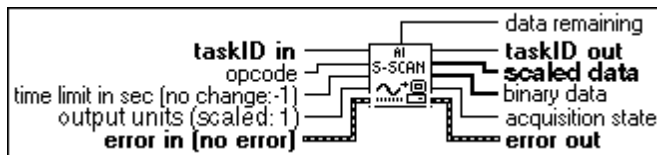
Reads data from a buffered data acquisition.



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

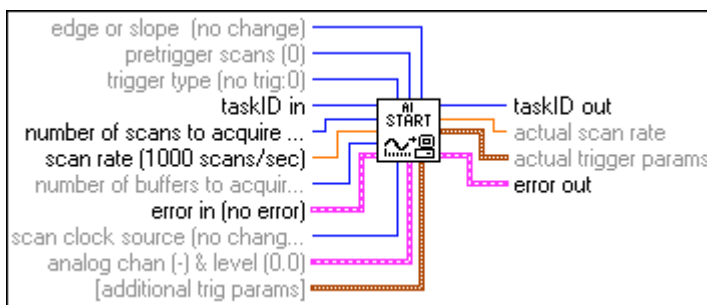


Note *LabVIEW restarts the device in the event of a FIFO overflow during a timed, nonbuffered acquisition.*

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.

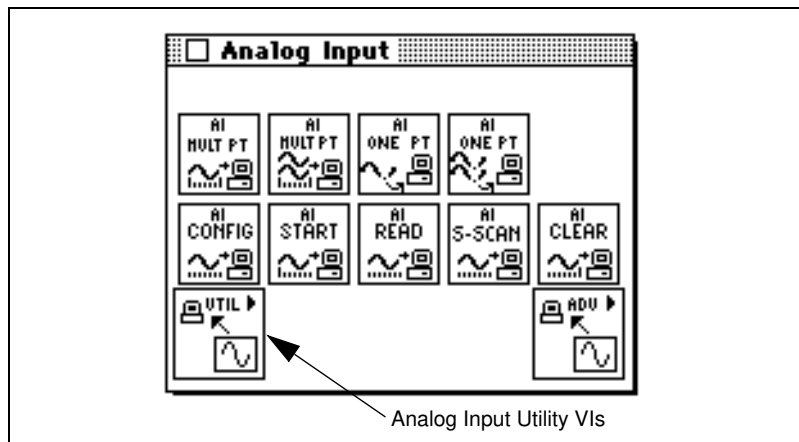


Refer to Appendix B, *DAQ Hardware Capabilities*, for the channel ranges, input limits, scanning order, triggers, and clocks you can use with your National Instruments DAQ device.

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 18, *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.



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.

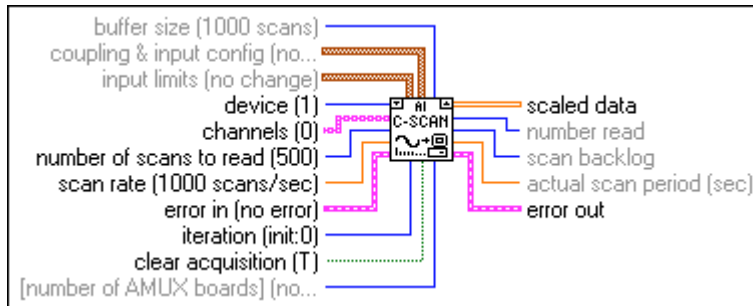
The General Error Handler VI is in **Functions»Time and Dialog** in LabVIEW. For more information on this VI, refer to Chapter 10, [Time, Dialog, and Error Functions](#).

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.

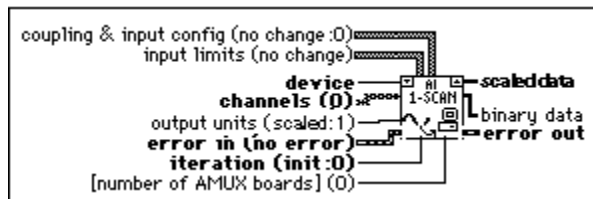
Also wire the condition that terminates the loop to the **clear acquisition** input, inverting the signal if necessary so that it reads TRUE on the last iteration. On iteration 0, the VI calls the AI Config VI to configure the channel group and hardware and allocates a data buffer; the VI calls the AI 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 scaled values. 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 B, *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 scaled or binary values.



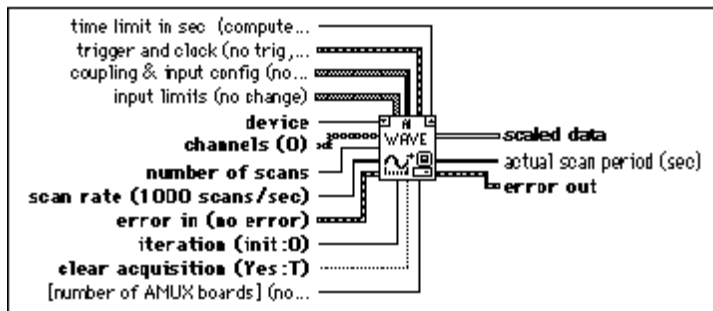
The AI Read One Scan VI performs an immediate measurement of a group of one or more channels. If you place the VI in a loop to take multiple measurements from a group of channels, wire the loop iteration terminal to the VI **iteration** parameter.

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 B, *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.



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 scaled values. 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 B, *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.

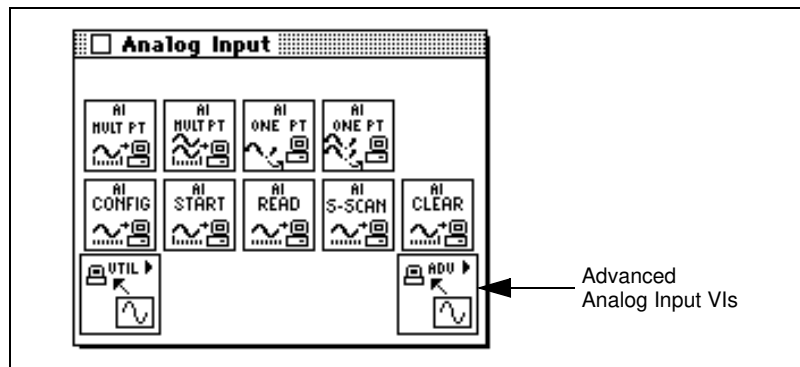
**Note**

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 .

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.

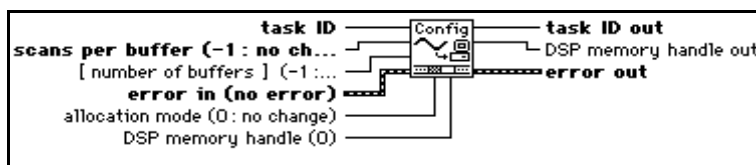


Advanced Analog Input VI Descriptions

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 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.

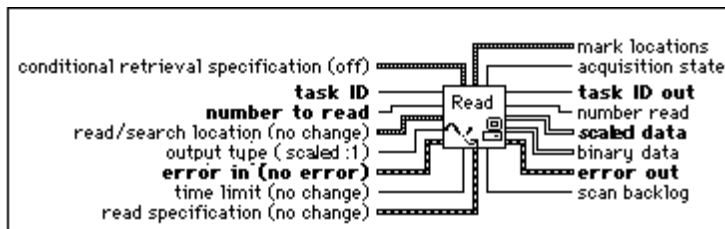
Table 18-1 lists default settings and ranges for the AI Buffer Config VI. The first row gives the values for most devices, and the other rows give the values for devices that are exceptions to the rule.

Table 18-1. 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
NB-A2000 NB-A2100 NB-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

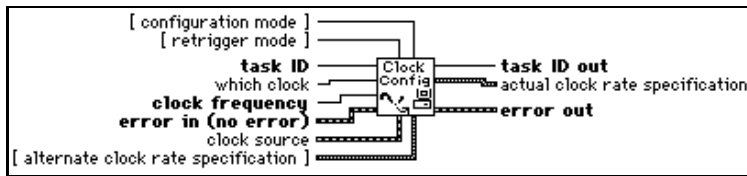
Returns analog input data from the internal data buffer(s).



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 B, *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, and NB-A2150), 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 18-2 lists default settings and ranges for the controls of the AI Clock Config VI.

Table 18-2. Device-Specific Settings and Ranges for Controls in the AI Clock Config VI

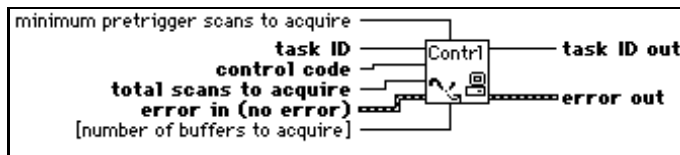
Device	Configuration Mode		Retrigger Mode	Which Clock		Clock Source	
	Default Setting	Range	Default Setting	Default Setting	Range	Default Setting	Range
AT-MIO-16E1 AT-MIO-16E-2 AT-MIO-64E-1 NEC-MIO-16E-4 PCI-MIO-16E-1 PCI-MIO-16E-4 PCI-MIO-16XE-10	1	1, 3	no support	1	1, 2	1	1, 2 $4 \leq n \leq 11$
PCI-6110E PCI-6111E	1	1, 3	no support	1	1	1	1, 2 $4 \leq n \leq 11$
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 $4 \leq n \leq 9$
NB-A2150 NB-A2100 NB-A2000	1	1, 3	no support	1	1	1	$1 \leq n \leq 3$
DSA Devices	1	1, 3	no support	1	1	1	1
PC-LPM-16 DAQCard-500 DAQCard-516 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
5102	1	1, 3	no support	1	1	1	1, 6

Table 18-2. Device-Specific Settings and Ranges for Controls in the AI Clock Config VI (Continued)

Device	Configuration Mode		Retrigger Mode	Which Clock		Clock Source	
	Default Setting	Range	Default Setting	Default Setting	Range	Default Setting	Range
5911, 5912	1	1, 3	no support	1	1,2	1	$1 \leq n \leq 3$
All Other Devices	1	1, 3	no support	1	1, 2	1	$1 \leq n \leq 3$

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 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
- (Macintosh) NB-A2100
- (Macintosh) NB-A2150

Table 18-3 lists default settings and ranges for the AI Control VI.

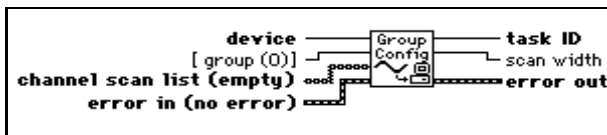
Table 18-3. Device-Specific Settings and Ranges for the AI Control VI

Device	Control Code		Total Scans to Acquire		Minimum Pretrigger Scans to Acquire		Number of Buffers to Acquire	
	DS*	R*	DS*	R*	DS*	R*	DS*	R*
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
5911, 5912	0	0, 4	0	n≥1	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 B, *DAQ Hardware Capabilities*, for the channel ranges and scanning order available with your DAQ device.

Table 18-4 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 18-4. 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	$0 \leq n \leq 15$	all channels	$0 \leq n \leq 15$
Most Macintosh Devices	0	$0 \leq n \leq 15$	all channels	$0 \leq n \leq 15$
AT-MIO-64F-5 AT-MIO-64E-1*	0	$0 \leq n \leq 15$	all channels	$0 \leq n \leq 63$
Lab-PC+, PCI-1200, DAQCard-1200	0	$0 \leq n \leq 15$	all channels	$0 \leq n \leq 7$
Lab-LC, Lab-NB	0	$0 \leq n \leq 15$	all channels	$0 \leq n \leq 7$
NB-A2000, NB-A2150	0	$0 \leq n \leq 15$	all channels	$0 \leq n \leq 3$
NB-A2100	0	$0 \leq n \leq 15$	all channels	0, 1
5102 Devices	0	$0 \leq n \leq 15$	all channels	0, 1
PCI-4452, PCI-4451	0	$0 \leq n \leq 15$	all channels	0, 1
PCI-4452, PCI-4552	0	$0 \leq n \leq 15$	all channels	$0 \leq n \leq 3$

* The valid channels for the AT-MIO-64E-1 in Differential Mode are 0-7, 16-23, 34-39, 48-55.



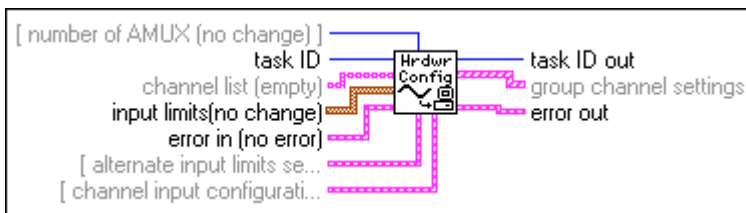
Note *The Lab-LC, Lab-NB, Lab-PC+, PCI-1200, PC-LPM-16, DAQCard-500, DAQCard-700, and DAQCard-1200 must scan channel lists containing multiple channels from channel n ($n \geq 0$) to channel 0 in sequential order, including all channels between n and 0. The NB-A2000, NB-A2150, EISA-A2000, and AT-A2150 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 B, 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 18-5 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 18-5. 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 .
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
```

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 B, *DAQ Hardware Capabilities*, for the channel ranges, input limits, and scanning order available with your DAQ device.

Tables 18-6 through 18-9 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 gives 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 18-6. 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	$1 \leq n \leq 3$	0	$0 \leq n \leq 4$	empty
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	$1 \leq n \leq 3$	0	0	empty
AT-MIO-16X, AT-MIO-64F-5	1	1	1 (no ~)	$1 \leq n \leq 3$	0	$0 \leq n \leq 4$	empty
NB-A2100, NB-A2150	1	1, 2	2	2	0	0	empty
DAQCard-500, DAQCard-516, DAQCard-700	1	1	2	1, 2	0	0	empty

Table 18-6. Device-Specific Settings and Ranges for the AI Hardware Config VI (Continued)

Device	Channel Input Configuration Cluster				Number of AMUX		Channel List
	Coupling		Input Mode ~				
	DS*	R*	DS*	R*	DS*	R*	DS*
5102 Devices	1	1, 2	2	2	0	0	empty
PCI-6110E, PCI-6111E, PCI-4451, PCI-4551, PCI-4452, PCI-4552	1	1, 2	1	1	0	0	empty

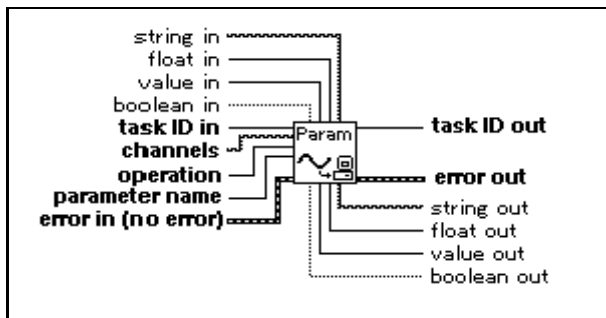
* DS = Default Setting; R = Range



Note Channels 0 and 1 and channels 2 and 3 must have the same coupling for the NB-A2150.

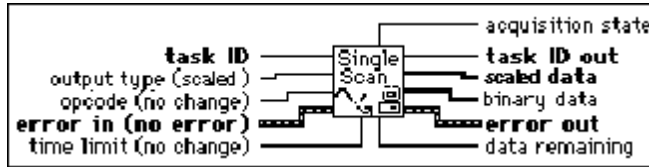
AI Parameter

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, DSA, and 59xx 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



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

Table 18-7 lists default settings and ranges for the AI SingleScan VI.

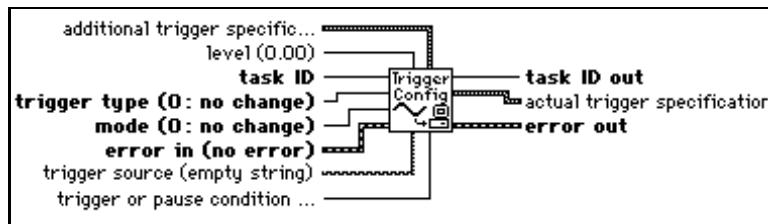
Table 18-7. Device-Specific Settings and Ranges for the AI SingleScan VI

Device	Output Type		Opcode		Time Limit	
	DS	R	DS	R	DS	R
NB-A2000, NB-A2100, NB-A2150	1	$1 \leq n \leq 3$	1	1	variable	$n \geq 0$
All Other Devices	1	$1 \leq n \leq 3$	1	$1 \leq n \leq 4$	$1 \leq n \leq 4$	$n \geq 0$

* DS = Default Setting; R = Range

AI Trigger Config

Configures the trigger conditions for starting the scan and channel clocks and the scan counter.



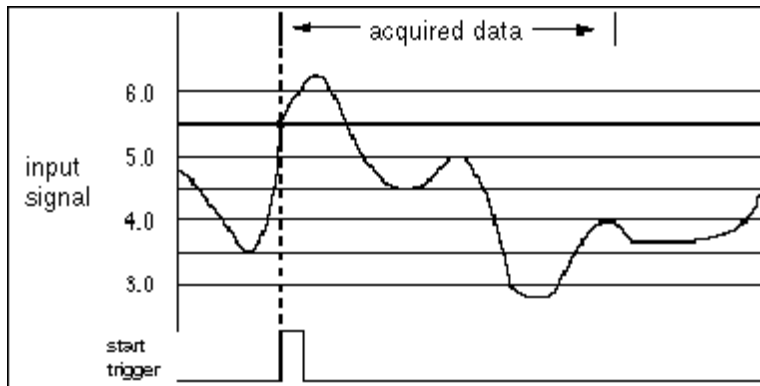
Refer to Appendix B, *DAQ Hardware Capabilities*, for information on the triggers available with your DAQ device. Refer to your E Series device user manual for a detailed description of the triggering capabilities of the device.

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 18-10, 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.

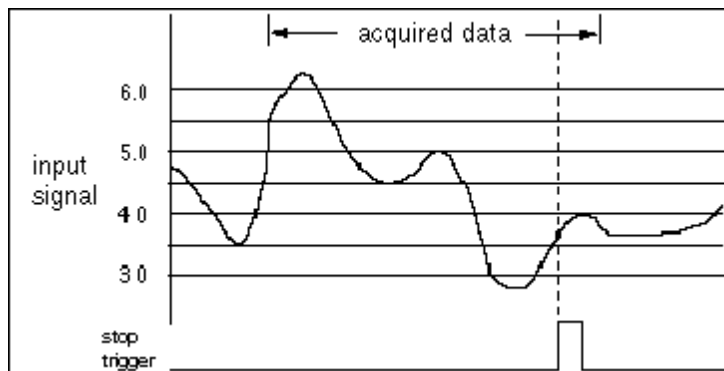
With E-Series devices, if you are using an analog trigger and the analog signal is connected to one of the analog input channels, that channel must be first in the scan list. This restriction does not apply if you connect the analog signal to PFI0.



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 18-10, 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.



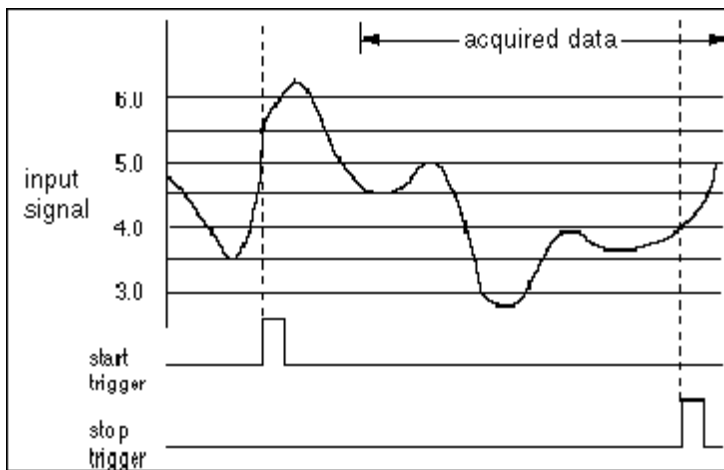
In the previous illustration, **total scans to acquire** is 1000 and **pretrigger scans to acquire** is 900. The stop trigger can come from digital trigger A or an analog trigger (**trigger or pause condition** = 1: Trigger on rising edge or slope, **level** = 3.7, **window size** = 0.5).

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 Tables 18-11 and 18-12 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 18-8.

Table 18-8. 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 18-9 shows the pin names on the I/O connector to which you should connect your digital trigger signal.

Table 18-9. 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
PC-LPM-16, DAQCard-500, DAQCard-700	no support	no support	no support
NB-A2000, NB-A2100, NB-A2150	EXTTRIG*	no support	EXTTRIG*
DSA 45xx	EXTTRIG*	EXTTRIG*	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 18-10 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 18-10. 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	$1 \leq n \leq 3$	no support		no support	
AT-MIO-16E-10, AT-MIO-16DE-10, AT-MIO-16XE-50, PCI-MIO-16XE-50	2	$2 \leq n \leq 4$	1	$1 \leq n \leq 3$	1	1, 2, 7, 8	no support	
AT-MIO-16E-2, AT-MIO-64E-3, NEC-MIO-16E-4	2	$1 \leq n \leq 6$	1	$1 \leq n \leq 3$	1	$1 \leq n \leq 8$	0	$-10 \leq n \leq 10$
Lab and 1200 Series devices	2	2	1	$1 \leq n \leq 3$	no support		no support	
PC-LPM-16, DAQCard-500, DAQCard-700	no support		no support		no support		no support	
NB-A2100, NB-A2150	1	1, 2	1	$1 \leq n \leq 3$	1	1, 2	0	$-2.828 \leq n \leq 2.828$
NB-A2000	1	1, 2	1	$1 \leq n \leq 3$	1	1, 2	0	$-5.12 \leq n \leq 5.12$
5102 Devices	1	1, 2, 3, 6	1	$1 \leq n \leq 3$	1	1, 2, 3, 4	0	$-5 \leq n \leq 5$
5911, 5912	1	1, 2, 3, 6	1	$1 \leq n \leq 3$	1	1, 2, 3, 4	0	$-10 \leq n \leq 10$
DSA Devices	1	1, 2, 3	1	$1 \leq n \leq 3$	1	$1 \leq n \leq 4$	0	$-42 \leq n \leq 42$

* DS = Default Setting; R = Range

Table 18-11. 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-1 AT-MIO-16E-2 NEC-MIO-16E-4 PCI-MIO-16E-1 PCI-MIO-16E-1 PCI-MIO-16XE-10	0	$0 \leq n \leq 15$, PFIO	0	$0 \leq n \leq 20$	no support	
AT-MIO-64E-3	0	$0 \leq n \leq 63$, PFIO	0	$0 \leq n \leq 20$	no support	
NB-A2000	0	$0 \leq n \leq 3$	no support		2	1, 2
NB-A2100 NB-A2150	0	$0 \leq n \leq 3$	0	$0 \leq n \leq 5.656$	1	1, 2
5102 Devices	0	1, 1, TRIG	0	$0 \leq n \leq 10$	1	1, 2
PCI-6110E	0	$0 \leq n \leq 4$ PFIO	0	$0 \leq n \leq 80$	1	1, 2
PCI-6111E	0	$0 \leq n \leq 2$ PFIO	0	$0 \leq n \leq 80$	1	1, 2
4451, 4551	0	0, 1	0	$0 \leq n \leq 84$	no support	
4452, 4552	0	$0 \leq n \leq 4$	0	$0 \leq n \leq 84$	no support	
All Other Devices	no support		no support		no support	

Table 18-12. Device-Specific Settings and Ranges for the AI Trigger Config VI (Part 3)

Device	Trigger Source (Digital)	
	DS	R
E-Series Start Trigger	PFI0	PFI 0~9, RTSI 0~6, GPCTRO
E-Series Stop Trigger	PFI1	PFI 0~9, RTSI 0~6
E-Series Digital Scan Clock Gate	PFI0	PFI 0~9, RTSI 0~6
5102 Devices with RTSI, Start and Stop Triggers	PFI0	PFI 1-2, RTSI 0-6
5102 Devices without RTSI, Start and Stop Triggers	PFI0	PFI1-2
5911, 5912	PFI1	PFI 1-2, RTSI 0-6
DSA 44xx Start Trigger	PFI0	PFI0, PFI1, PFI3, PFI4, PFI6, RTSI 0~6
OSA 44xx Stop Trigger	PFI1	PFI0, PFI1, PFI3, PFI4, PFI6, RTSI 0~6
DSA 45xx Start and Stop Trigger	dedicated EXTTRIG* pin	PFI 0~33, RTSI 0~6
All Other Devices	no support*	

* See Table 18-9 for devices with fixed digital trigger sources.

Table 18-13. Device-Specific Settings and Ranges for the AI Trigger Config VI (Part 4)

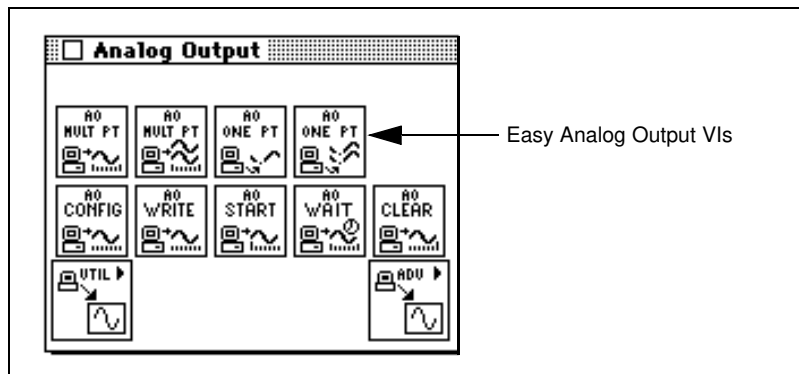
Device	Additional Trigger Specifications Cluster					
	Delay		Skip Count		Time Limit	
	DS	R	DS	R	DS	R
NB-A2000	0	$0 \leq n \leq 655.35$	no support		no support	
NB-A2100, NB-A2150S	0	$0 \leq n \leq 32.77$	no support		no support	
NB-A2150C	0	$0 \leq n \leq 16.38$	no support		no support	
NB-A2150F	0	$0 \leq n \leq 17.05$	no support		no support	
All Other Devices	no support		no support		no support	

* DS = Default Setting; R = Range

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 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 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 waveforms on the specified analog output channels at the specified update rate.

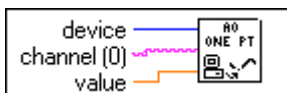


If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

Refer to Appendix B, *DAQ Hardware Capabilities*, for the channel numbers you can use with your DAQ device.

AO Update Channel

Writes a specified value to an analog output channel.

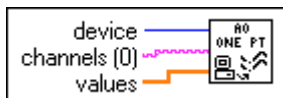


The AO Update Channel VI writes a single update to an analog output channel. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

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

AO Update Channels

Writes values to each of the specified analog output channels.



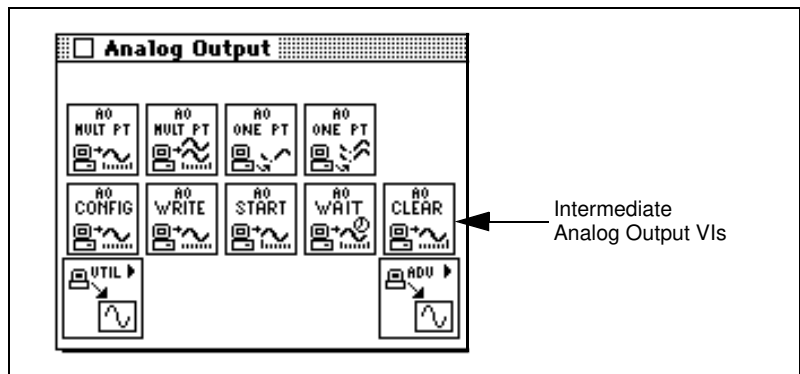
The AO Update Channels VI updates multiple analog output channels with single values. If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

Refer to Appendix B, *DAQ Hardware Capabilities*, for the channel numbers you can use with your DAQ device.

Intermediate Analog Output VIs

This chapter describes the Intermediate Analog Output VIs. These VIs 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 22, *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.



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.

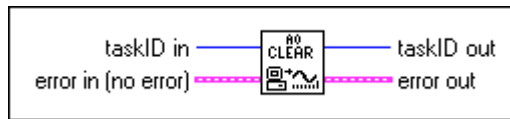
The General Error Handler VI is in **Functions»Time and Dialog** in LabVIEW. For more information on this VI, refer to your *LabVIEW User Manual*.

Analog Output VI Descriptions

The following Analog Output VIs are available.

AO Clear

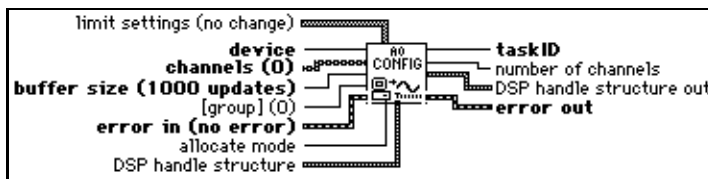
Clears the analog output task associated with **taskID in**.



The AO Clear VI always clears the generation regardless of whether **error in** indicates an error.

AO Config

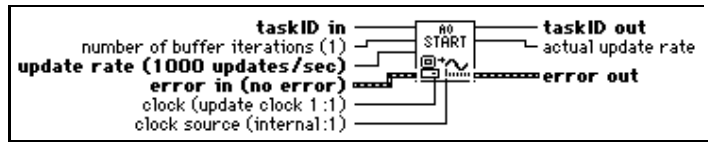
Configures the channel list and output limits, and allocates a buffer for analog output operation.



Refer to Appendix B, *DAQ Hardware Capabilities*, for the channel ranges and output limits available with your DAQ device.

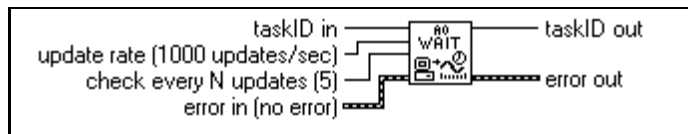
AO Start

Starts a buffered analog output operation. This VI sets the update rate and then starts the generation.



AO Wait

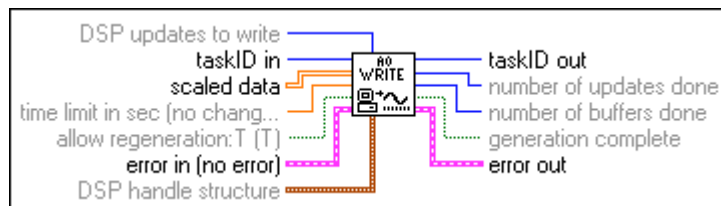
Waits until the waveform generation of the task completes before returning.



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

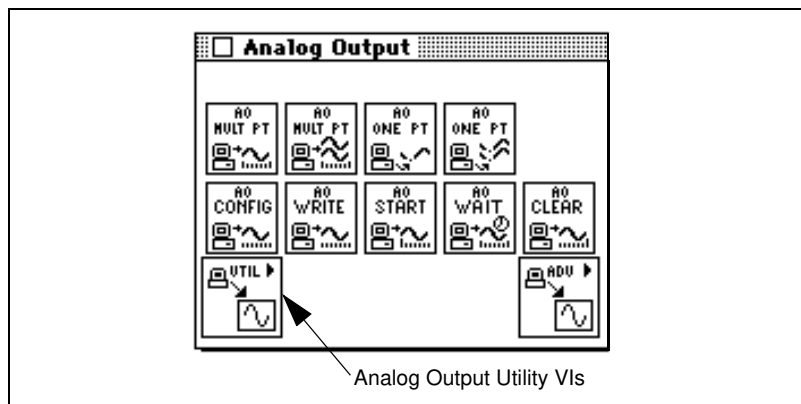
Writes data into the buffer for a buffered analog output operation.



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 22, *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.



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.

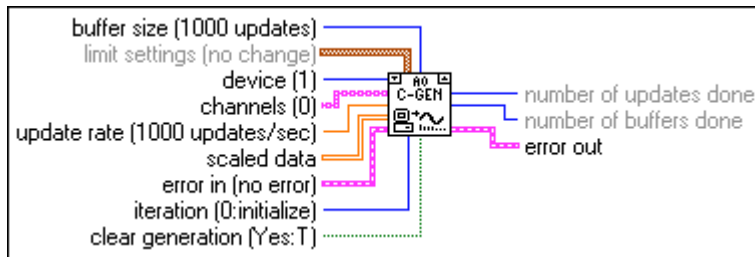
The General Error Handler VI is in **Functions»Utilities** in LabVIEW. For more information on this VI, refer to Chapter 10, *Time, Dialog, and Error Functions*.

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.



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 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.

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 B, *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 B, *DAQ Hardware Capabilities*, for the channel ranges, output limits, and scanning order available with your DAQ device.

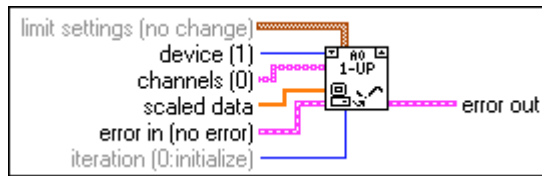


Note

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 value to each of the specified analog output channels.



i
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 B, [DAQ Hardware Capabilities](#), for the channel ranges, output limits, and scanning order available with your DAQ device.



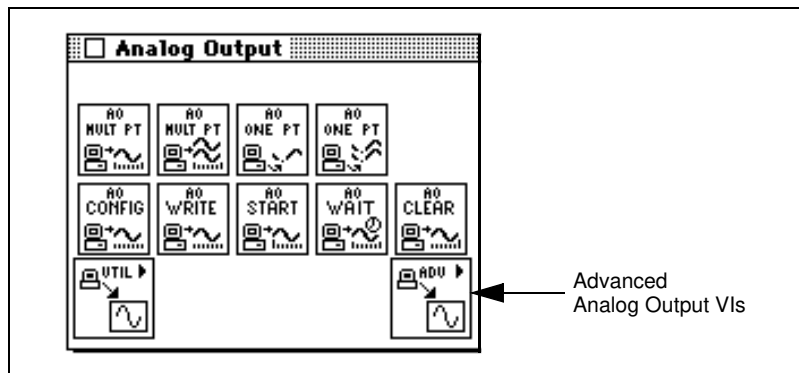
Note

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.

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.



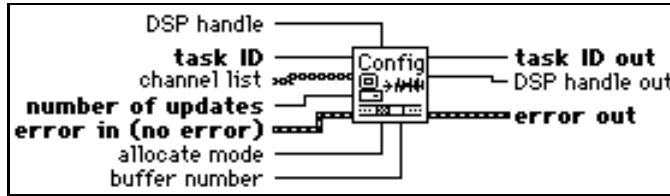
Advanced Analog Output VI Descriptions

The following Advanced Analog Output VIs are available.

AO Buffer Config

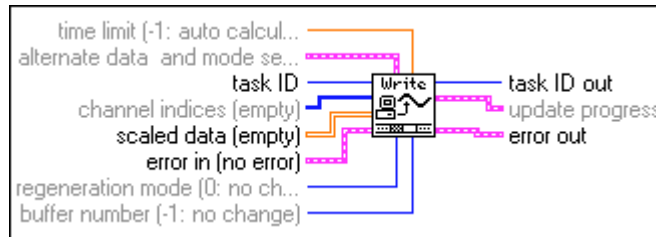
Allocates memory for an analog output buffer. If you are using interrupts, you can allocate a series of analog output buffers and assign them to a group by calling the AO Buffer Config VI multiple times. Each buffer can have its own size. If you are using DMA, you may allocate only one buffer.

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.

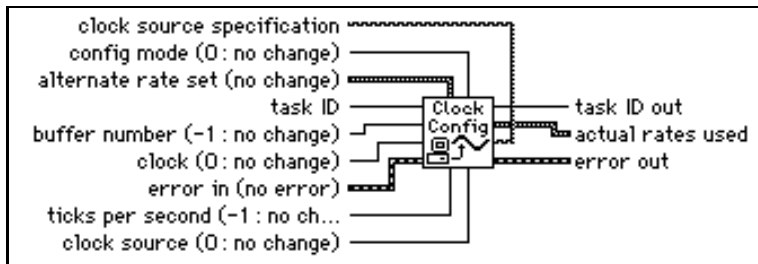


You wire the new data to one of three inputs—**scaled data**, **binary data**, or **DSP memory handle**. The VI searches these inputs in that order for the first array with a length greater than zero. The VI then writes the data from this array to the output buffer. The length of the **scaled data** 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.

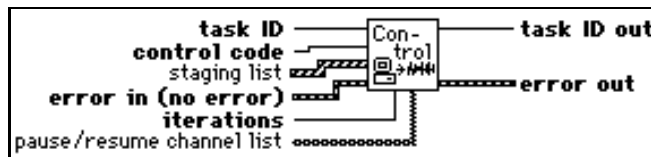


Refer to Appendix B, *DAQ Hardware Capabilities*, for the clocks available with your DAQ device.

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.

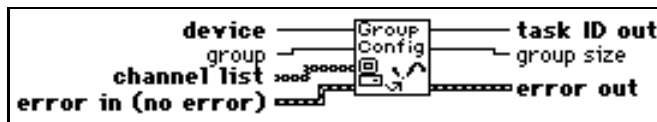
AO Control

Starts, pauses, resumes, and clears analog output tasks.



AO Group Config

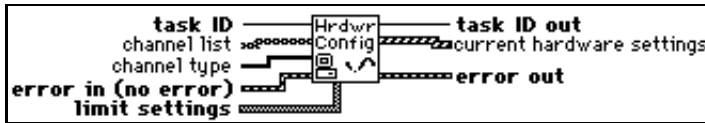
Assigns a list of analog output channels to a group number and produces the taskID that all the other analog output VIs use.



Refer to Appendix B, *DAQ Hardware Capabilities*, for the channels available with your DAQ device.

AO Hardware Config

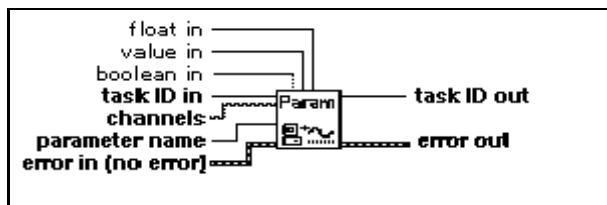
Configures the limits (polarity and reference) and whether data for a given channel is expressed in volts milliamperes if you are using channel numbers. This VI always returns the current settings for all the channels in the group.



Refer to Appendix B, *DAQ Hardware Capabilities*, for the channel ranges, and output limits available with your DAQ device.

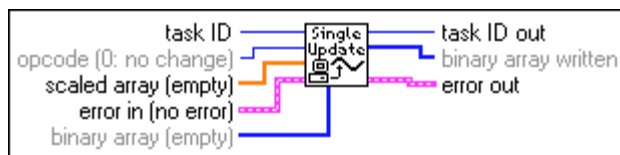
AO Parameter

Sets miscellaneous parameters associated with the Analog Output operation of the devices that are not covered with other Analog Output VIs.



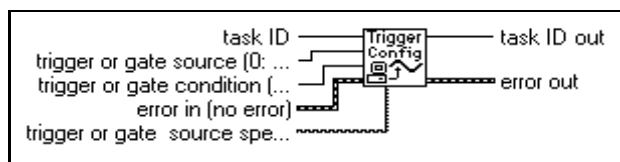
AO Single Update

Performs an immediate update of the channels in the group.



AO Trigger and Gate Config (Windows)

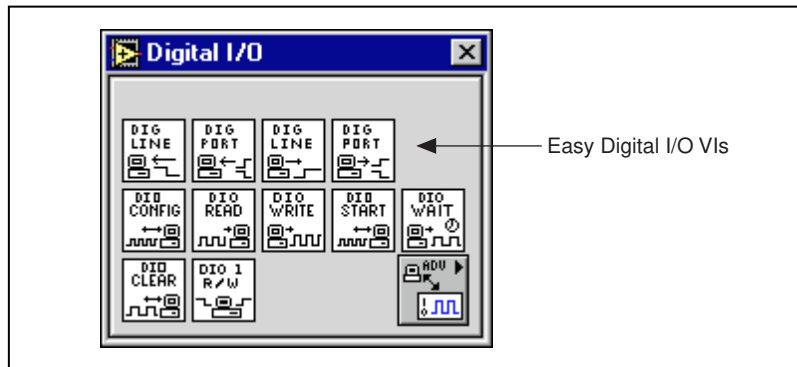
Configures the trigger and gate conditions for analog output operations on E-Series devices and 5411 devices.



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**.



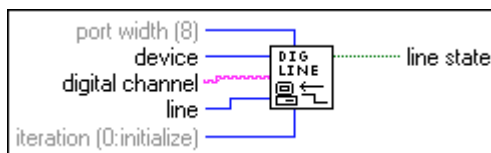
The Easy Digital I/O VIs are the VIs on the top row of the **Digital I/O** palette. For examples of how to use the Easy Digital I/O VIs, open the example library by opening `examples\daq\digital\digital.lib`.

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 digital channel that you configure.



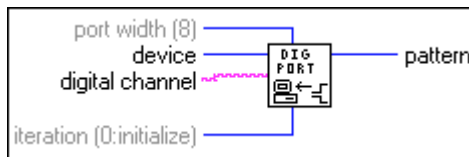
If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

**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 channel that you configure.



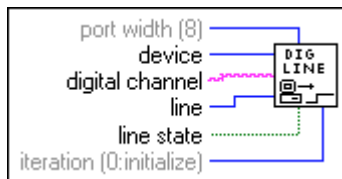
If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

**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 channel that you specify.



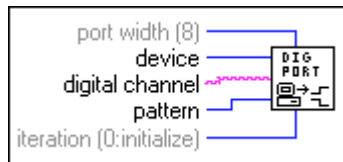
If an error occurs, a dialog box appears, giving you the option to stop the VI or continue.

**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 channel 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.

Intermediate Digital I/O VIs

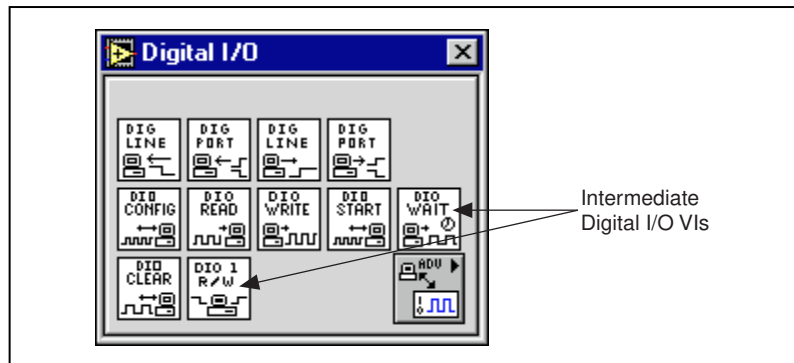
This chapter describes the Intermediate Digital I/O VIs. These VIs are single VI solutions to common digital problems.

For example, the DIO Single Read/Write VI is a single VI solution for non-buffered digital reads and writes. 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 these VIs.

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.

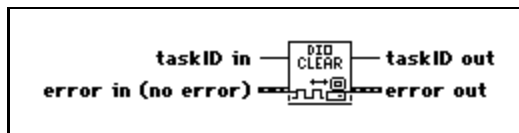
The General Error Handler VI is in **Functions»Time and Function** in LabVIEW. For more information on this VI, refer to Chapter 10, [Time, Dialog, and Error Functions](#).

Intermediate Digital I/O VI Descriptions

The following Intermediate Digital I/O VIs are available.

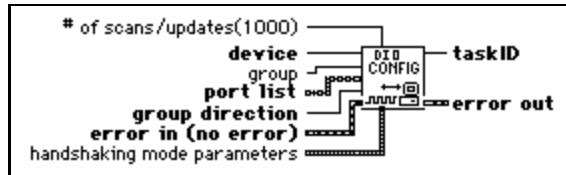
DIO Clear

Calls the Digital Group Buffer Control VI to halt a transfer and clear the group.



DIO Config

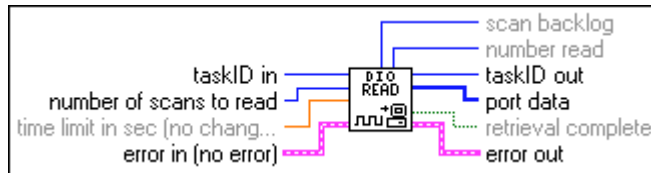
The DIO Config VI calls the advanced Digital Group Config VI to assign a list of ports to the group, establish the group's direction, and produce the **taskID**. The VI then calls the Digital Mode Config VI to establish the handshake parameters, which only affect the operation of the DIO-32 devices. 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 B, *DAQ Hardware Capabilities*, for the ports and directions available with your DAQ device.

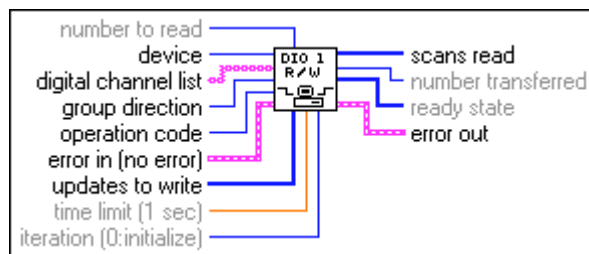
DIO Read

Calls the Digital Buffer Read VI to read data from the internal transfer buffer and returns the data read in **pattern**.



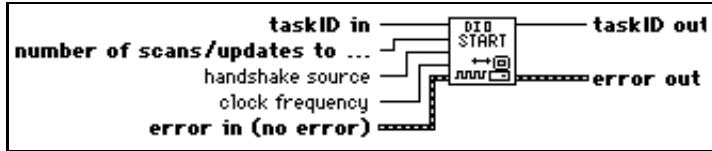
DIO Single Read/Write

Reads or writes digital data to the digital channels specified in the digital channel 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 configuration takes place only once.



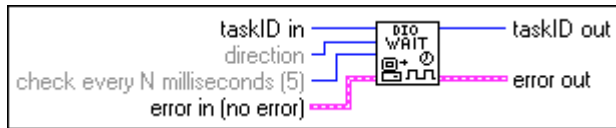
DIO Start

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.

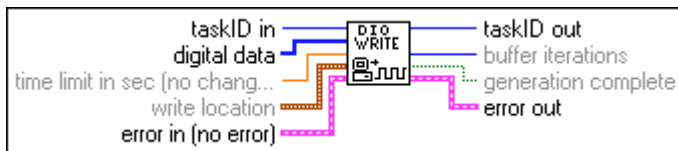


Refer to Appendix B, *DAQ Hardware Capabilities*, for the handshake modes available with your DAQ device.

DIO Write

Calls the Digital Buffer Write VI to write to the internal transfer buffer.

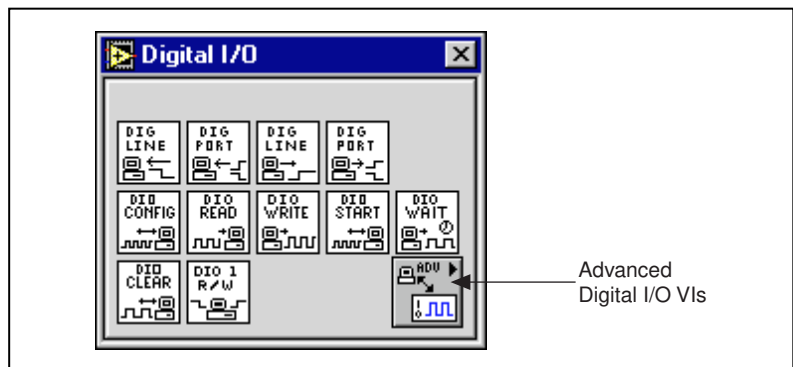
(Macintosh) You must fill the buffer with data before you use the DIO Start VI to begin the digital output operation. You can call the DIO Write VI after the transfer begins to retrieve status information.



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.

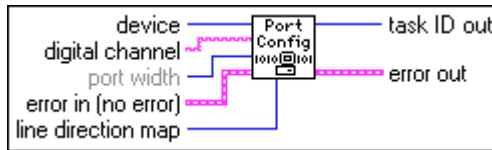


Digital Port VI Descriptions

The digital port VIs perform immediate digital reads and writes only.

DIO Port Config

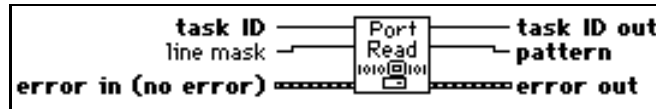
Establishes a digital channel configuration. You can use the **taskID** that this VI returns only in digital port VIs.



Refer to Appendix B, *DAQ Hardware Capabilities*, for the ports and directions available with your DAQ device.

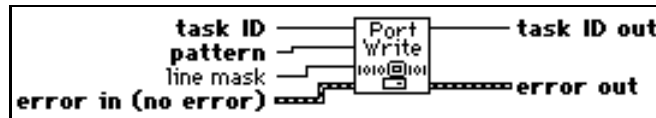
DIO Port Read

Reads the digital channel identified by **taskID** and returns the pattern read in **pattern**.



DIO Port Write

Writes the value in **pattern** to the digital port identified by **taskID**.

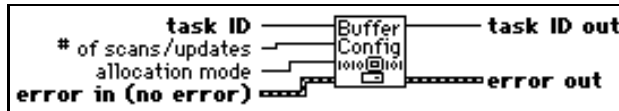


Digital Group VI Descriptions

The digital group VIs perform immediate, handshaked, or clocked digital I/O.

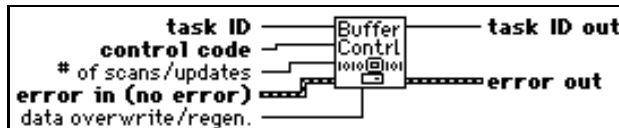
Digital Buffer Config

Allocates memory for a digital input or output buffer.



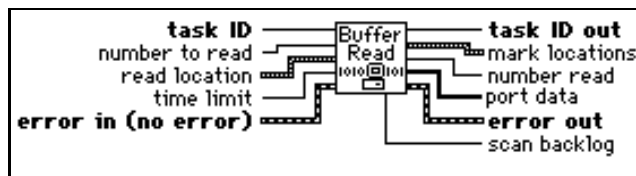
Digital Buffer Control

Starts an input or output operation.



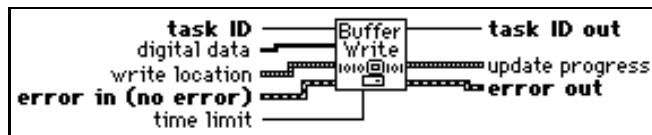
Digital Buffer Read

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.

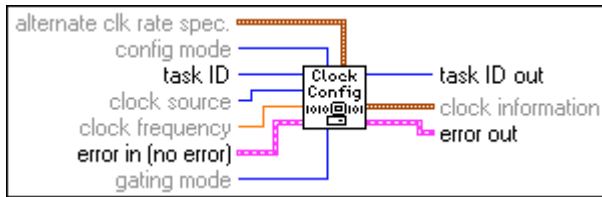


(Macintosh) Fill the buffer with data before you use the Digital Buffer Control VI to begin the digital output operation. You can call the Digital Buffer Write VI after the transfer begins to retrieve status information.

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 B, *DAQ Hardware Capabilities*, for the clocks available with your DAQ device.

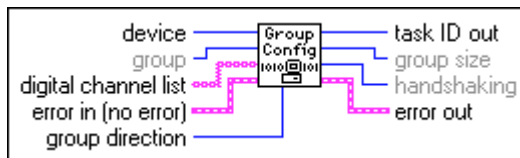
The following example illustrates how to use the three timebase parameters to specify a clock rate. Assume these parameters have the following settings:

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 B, *DAQ Hardware Capabilities*, for the ports and directions available with your DAQ device.



Note

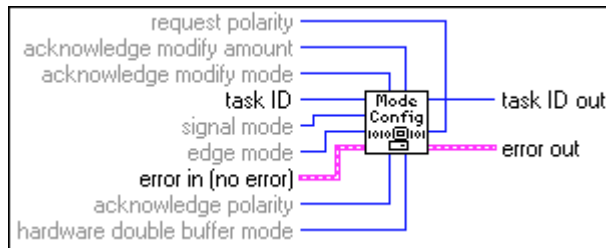
The same digital channel cannot belong to two different groups. If you configure a group to use a specified digital channel, that digital channel must be one that is not already defined in another group or you will get an error.

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. Handshaking is not allowed if **digital channel list** is composed of channel names. The AT-MIO-16D and AT-MIO-16DE-10 do not allow handshaking if **digital channel list** includes ports 0, 1, and/or 4. The DIO-96 devices do not allow handshaking if **digital channel list** includes ports 2, 5, 8, and/or 11. The DIO-24 and Lab and 1200 Series devices do not allow handshaking if **digital channel 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 B, *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.

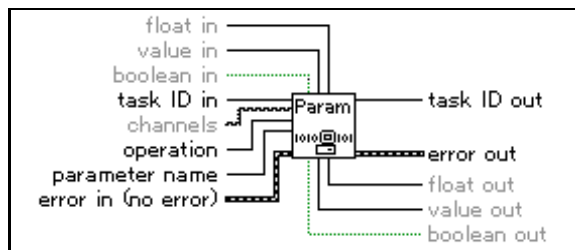


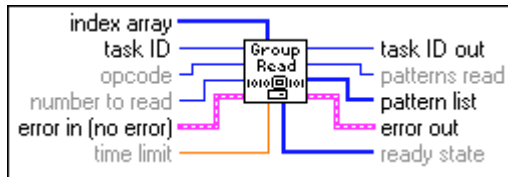
Table 25-1 lists device specific parameters and legal ranges for devices.

Table 25-1. 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
DAQ-DIO-653 (DIO- 32HS)	1: ACK/Req Exchange	per group	yes	taskID in, value in, value out	Off, On	N/A
	2: Clock Reverse	per group	yes	taskID in, value in, value out	Off, On	N/A

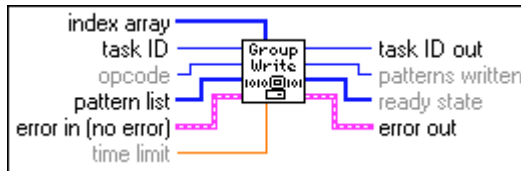
Digital Single Read

Reads the digital channels that belong to the group identified by **taskID** and returns the patterns read.



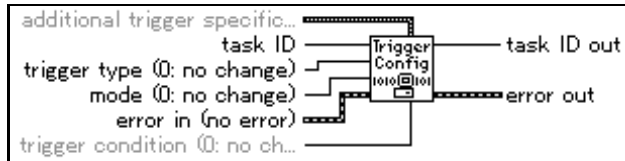
Digital Single Write

Writes the data in **pattern array** to the digital channels that belong to the group identified by **taskID**.



Digital Trigger Config

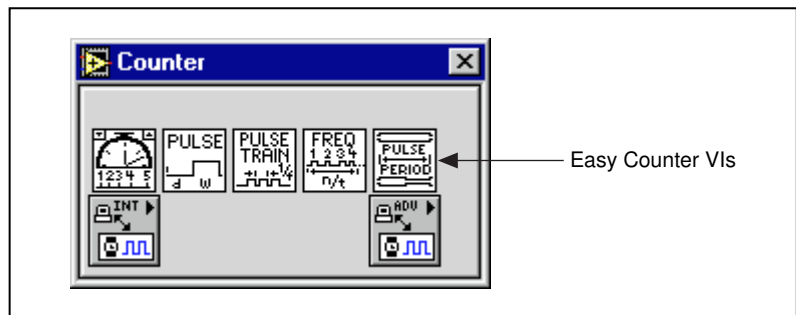
Configures the trigger condition for starting and/or stopping a digital pattern generation operation. This VI is only valid when the Digital Clock Config VI has its **handshake source** parameter set to 1 or 4 (internal or external pattern generation with external clock).



Easy Counter VIs

This chapter describes the Easy Counter VIs that 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 DAQ-STC or Am9513 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 8253/54 chip. Use the intermediate-level ICTR Control for those devices. Refer to Chapter 27, [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 27, *Intermediate Counter VIs*, for more information.

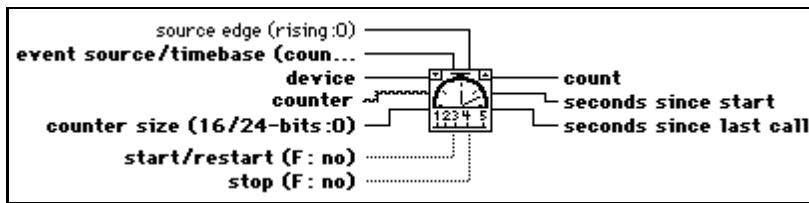
For examples of how to use the Easy Counter VIs, open the example libraries located in `examples\daq\counter`.

Easy Counter VI Descriptions

The following Easy Counter VIs are available.

Count Events or Time

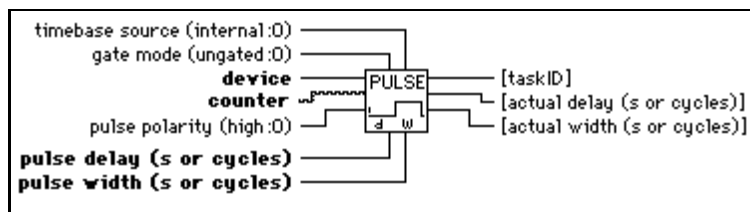
Configures one or two counters to count external events or elapsed time. An external event is a high or low signal transition on the specified SOURCE pin of the counter.



To count events, set **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.

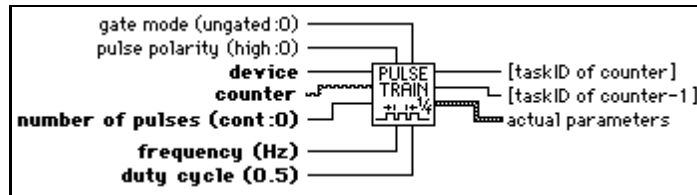
Generate Delayed Pulse

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 then 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.



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 consists of a delay phase (phase 1) followed by a pulse phase (phase 2).



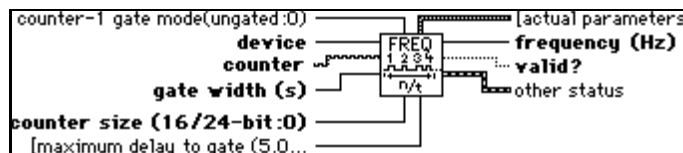
This VI uses only the specified **counter** to generate a continuous pulse. For a finite-length pulse, the VI also uses **counter-1** to generate a minimum-delayed pulse to gate **counter**. To generate another pulse train, execute the intermediate Counter Start VI with the **taskIDs** supplied by this 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 **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.*

Measure Frequency

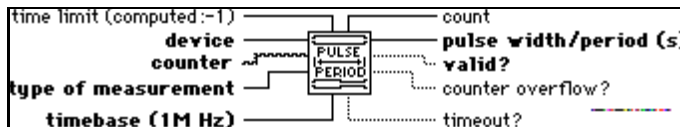
Measures the frequency of a TTL signal on the specified counter's SOURCE pin by counting positive edges of the signal during a specified period of time. In addition to this connection, you must wire **counter**'s GATE pin to the OUT pin of **counter-1**. This VI is useful for relatively high frequency 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 for Am9513) 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 **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.



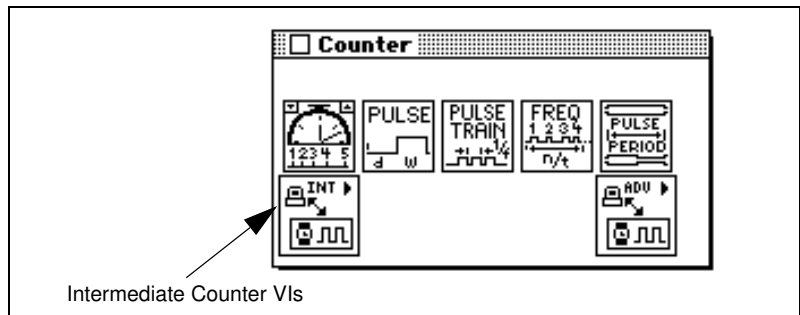
The VI iterates until a valid measurement, **timeout**, or **counter overflow** occurs. A valid measurement exists when **count** ≥ 4 without a counter overflow. If **counter overflow** occurs, lower **timebase**. If you start a pulse width measurement during the phase you want to measure, you get an incorrect low measurement. Therefore, make sure the pulse does not occur until after **counter** is started. This restriction does not apply to period measurements.

Intermediate Counter VIs

This chapter describes Intermediate Counter VIs you can use to program counters on MIO, TIO, and other devices with the DAQ-STC or Am9513 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 the Intermediate Counter 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 8253/54 counter/timer chip.

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



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.

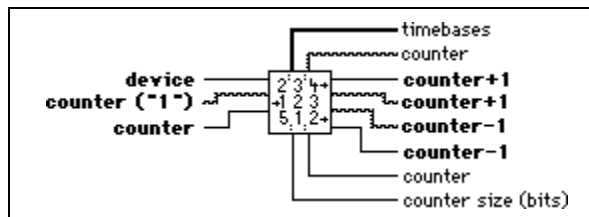
The General Error Handler VI is in **Functions»Utilities** in LabVIEW. For more information on this VI, refer to your *LabVIEW User Manual*.

Intermediate Counter VI Descriptions

The following Intermediate Counter VIs are available.

Adjacent Counters

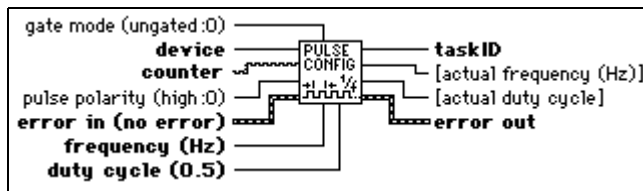
Identifies the counters logically adjacent to a specified counter of an MIO or TIO device. It also returns the counter size (number of bits) and the timebases.



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

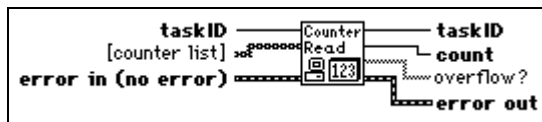
Configures a counter to generate a continuous TTL pulse train on its OUT pin.



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

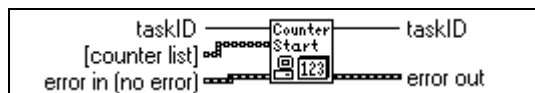
Reads the counter or counters identified by **taskID**.



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.

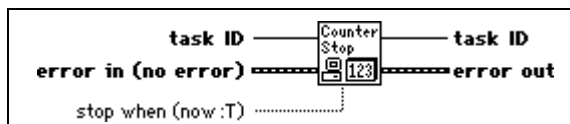
Counter Start

Starts the counters identified by **taskID**.



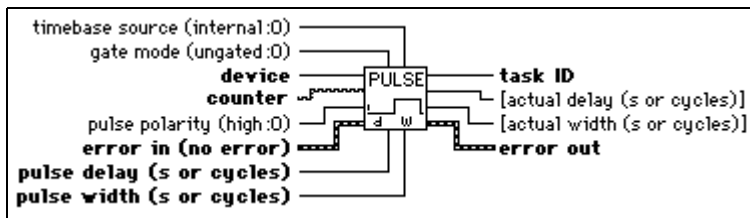
Counter Stop

Stops a count operation immediately or conditionally on an input error.



Delayed Pulse Generator Config

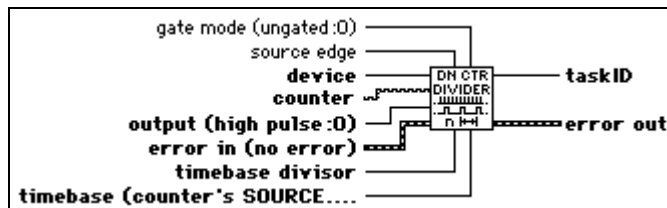
Configures a counter to generate a single, delayed TTL pulse on its OUT pin.



The signal is created by decrementing **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 **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 **counter**'s GATE pin. Call the Counter Start VI to start the pulse or enable it to be gated.

Down Counter or Divider Config

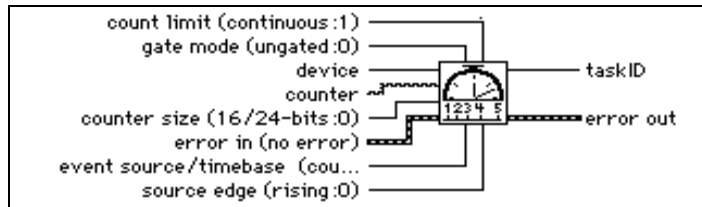
Configures the specified **counter** to count down or divide a signal on the counter's SOURCE pin or on an internal timebase signal using a count value called **timebase divisor**. The result is that the signal on the counter's OUT pin is equal to the frequency of the input signal divided by **timebase 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.

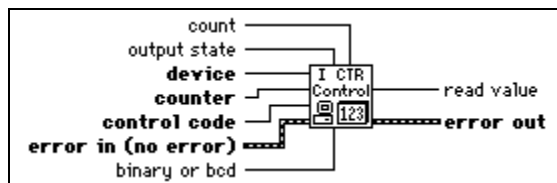


When the internal timebase is used, this VI works like the Tick Count (ms) function but uses a hardware counter on the DAQ device with programmable resolution. You can optionally gate or trigger the operation with a signal on the counter's GATE pin. Call the Counter Start VI to start the operation or enable it to be gated.

ICTR Control

Controls counters on devices that use the 8253/54 chip, including:

- Lab and 1200 Series devices, DAQCard-500, and DAQCard-700
- **(Windows)** LPM devices, 516 devices



In setup mode 0, as shown in Figure 27-1, the output becomes low after the mode set operation, and **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.

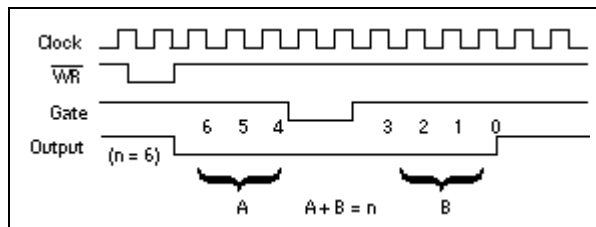


Figure 27-1. Setup Mode in ICTR Control

In setup mode 1, as shown in Figure 27-2, the output becomes low on **count** following the leading edge of the gate input and becomes high on TC.

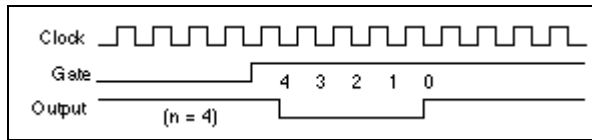


Figure 27-2. Setup Mode 1 in ICTR Control

In setup mode 2, as shown in Figure 27-3, the output becomes low for one period of the clock input. The **count** indicates the period between output pulses.

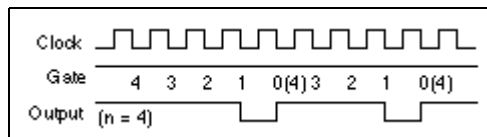


Figure 27-3. Setup Mode 2 in ICTR Control

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 27-4.

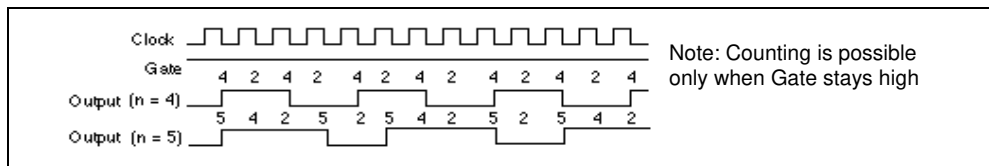


Figure 27-4. Setup Mode 3 in ICTR Control

In setup mode 4, as in Figure 27-5, the output is initially high, and **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.

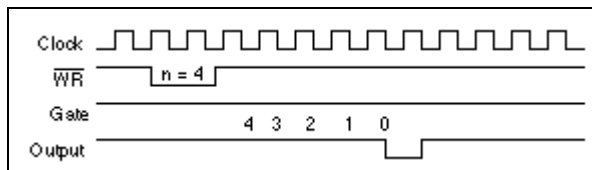


Figure 27-5. Setup Mode 4 in ICTR Control

Setup mode 5 is similar to mode 4, except that the gate input triggers the count to start. See Figure 27-6 for an illustration of mode 5.

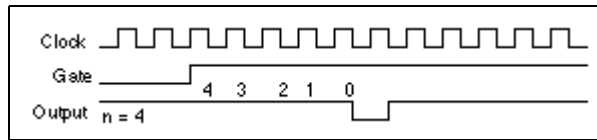
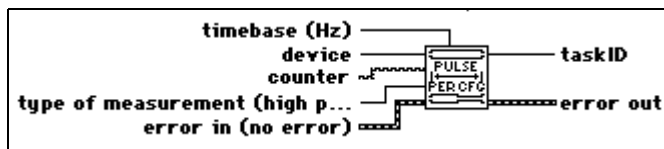


Figure 27-6. Setup Mode 5 in ICTR Control

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

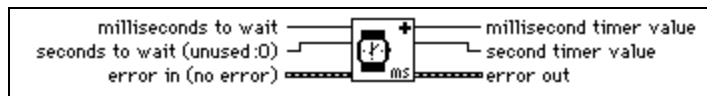
Configures the specified **counter** to measure the pulse width or period of a TTL signal connected to its GATE pin.



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 **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.

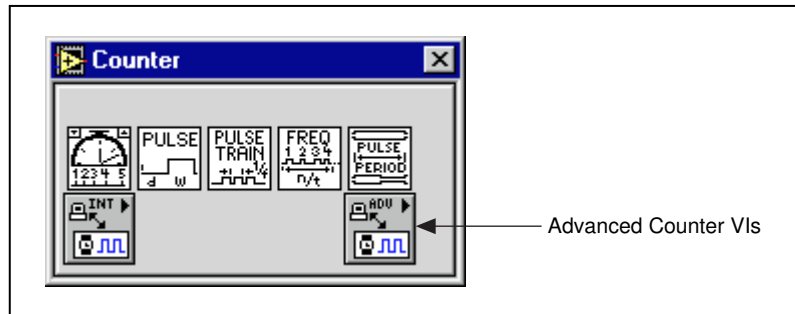


This VI is useful when you want to wait between calls to I/O subVIs that use the error I/O mechanism; without it you need to use a Sequence Structure to control the execution order.

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 or time, and to measure periods and frequencies.

You can access the **Advanced Counter** palette by choosing **Functions»Data Acquisition»Counter»Advanced Counter**. 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

Use only the inputs that you need on each VI when working with data acquisition. 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:

- DAQ-STC Counter Chip
- Am9513 Counter Chip
- 8253/54 Counter Chip

The ICTRControl VI is the only VI that works with devices that contain the 8253/54 counter chip.

Refer to Table 28-1 for the counter chips used with the various devices.

Table 28-1. Counter Chips and Their Available DAQ Devices

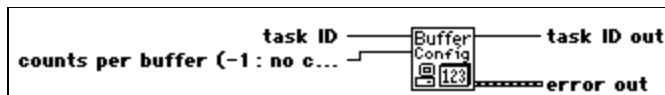
Counter Chip	DAQ Device
Am9513	AT-MIO-16, AT-MIO-16D, AT-MIO-16F-5, AT-MIO-16X, AT-MIO-64F-5, PC-TIO-10, All AO-2DC Devices, EISA-A2000, NB-MIO-16, NB-MIO-16X, NB-DMA-8-G, NB-DMA2800, NB-TIO-10, NB-A2000
DAQ-STC	All E Series Devices, 5102 Devices
8253/54	All Lab and 1200 Series Devices, DAQCard-500, DAQCard-700, LPM Devices, 516 Devices

Advanced Counter VI Descriptions

The following Advanced Counter VIs are available.

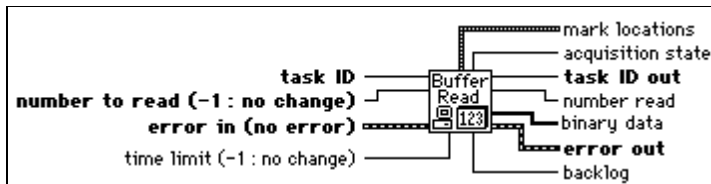
CTR Buffer Config

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 supported. However, circular use of the buffer is not implemented. Therefore, you must set up a finite buffer. You can read from the finite buffer as it fills.*

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 counters in the same group.

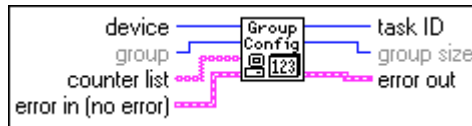


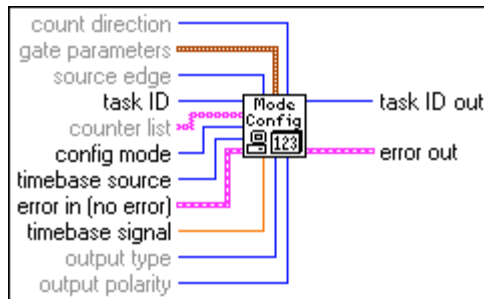
Table 28-2 contains valid counter numbers for devices supported by this VI.

Table 28-2. 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 **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 **gate mode** is set to high-level or rising-edge and **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 28-1 shows the count values you would read with this mode using two **gate mode** settings (high-level gating and rising-edge gating).

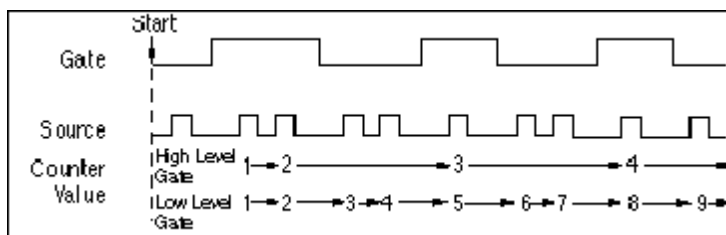


Figure 28-1. Unbuffered Mode 2 and 3 Counting

Use mode 3 to count transitions of the selected signal continuously, rolling over at TC and then continuing on. Figure 28-1 shows unbuffered mode 3 counting. Figure 28-2 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.

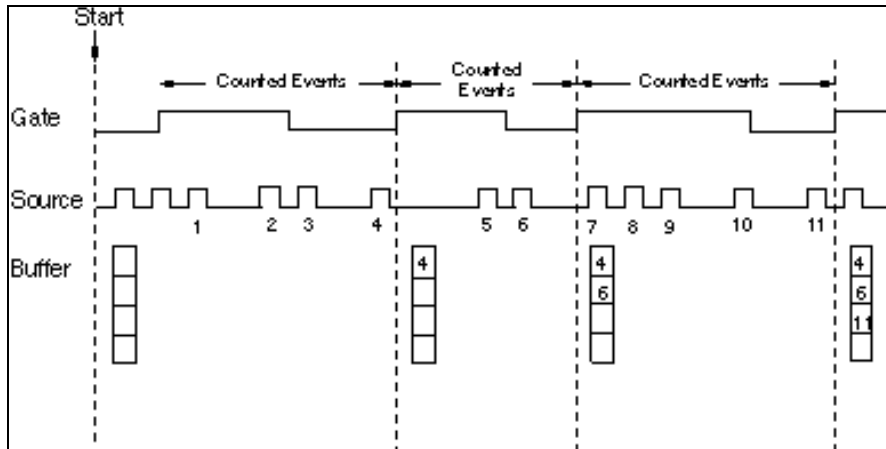


Figure 28-2. Buffered Mode 3 Counting

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 **gate mode** to high or low level. Figure 28-3 shows unbuffered mode 4 pulse width measurements. You can start an Am9513 counter at any time, and it measures 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.

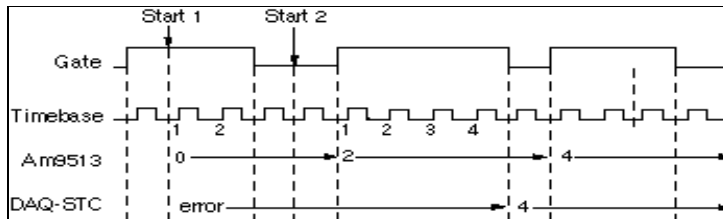


Figure 28-3. Unbuffered Mode 4 High Pulse Width Measurement

Figure 28-4 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.

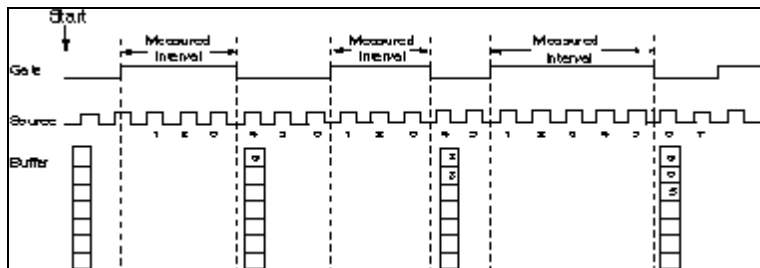


Figure 28-4. Buffered Mode 4 Rising-Edge Pulse Width Measurement

To measure period, set **gate mode** to rising or falling edge. Figure 28-5 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.

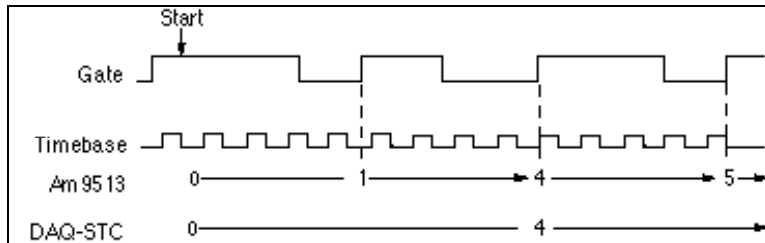


Figure 28-5. Unbuffered Mode 4 Rising-Edge Period Measurement

Figure 28-6 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.

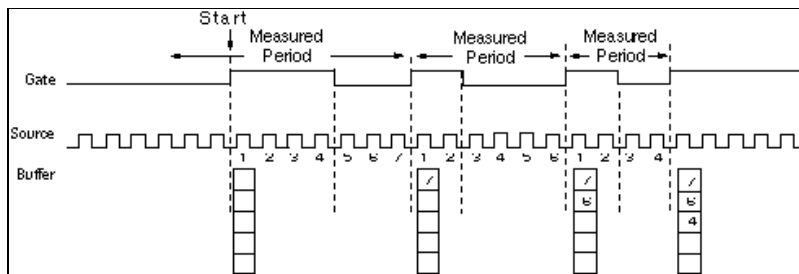


Figure 28-6. Buffered Mode 4 Rising-Edge Pulse Width Measurement

Use mode 5 to configure for pulse generation when you also need to configure **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 for 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 28-7.

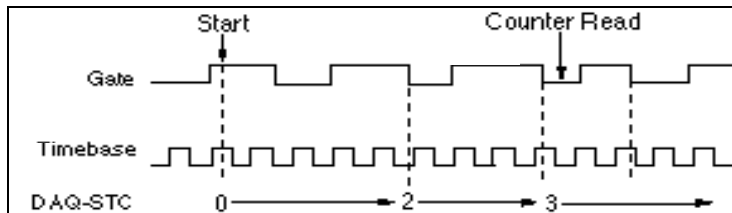


Figure 28-7. Unbuffered Mode 6 High Pulse Width Measurement

With buffered mode 6 counting, the measured value is stored into the buffer at the end of each pulse, as illustrated with Figure 28-8. Call the CTR Buffer Read VI to read the values.

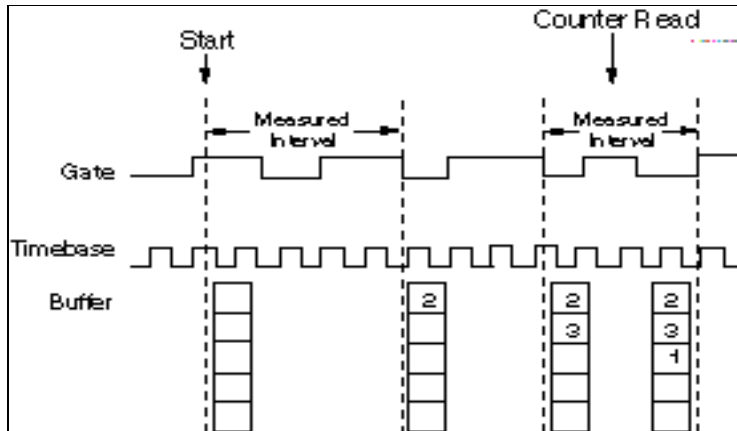


Figure 28-8. Buffered Mode 6 High Pulse Width Measurement (Count on Rising Edge of Source)

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.

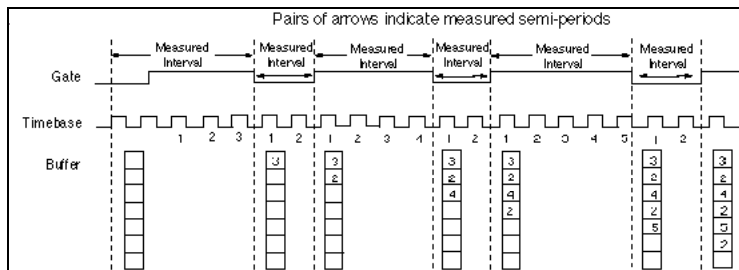


Figure 28-9. Buffered Mode 7 Semi-Period Measurement

Table 28-3 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.

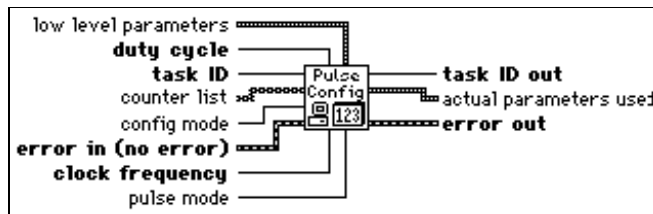
Refer to Table 28-3 to determine what is the next higher or lower consecutive counter.

Table 28-3. 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

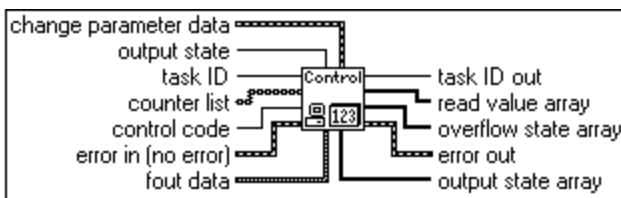
Specifies the parameters for pulse generation. This VI configures the counters but does not start them. Use the CTR Control VI with control code 1 (Start) to produce the pulse.



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.

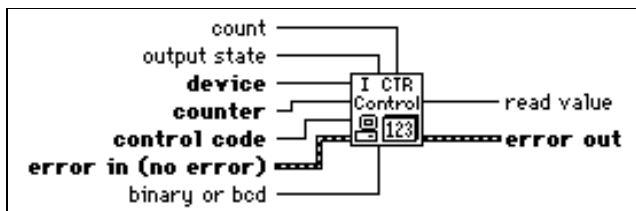
CTR Control

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).



Calibration and Configuration VIs

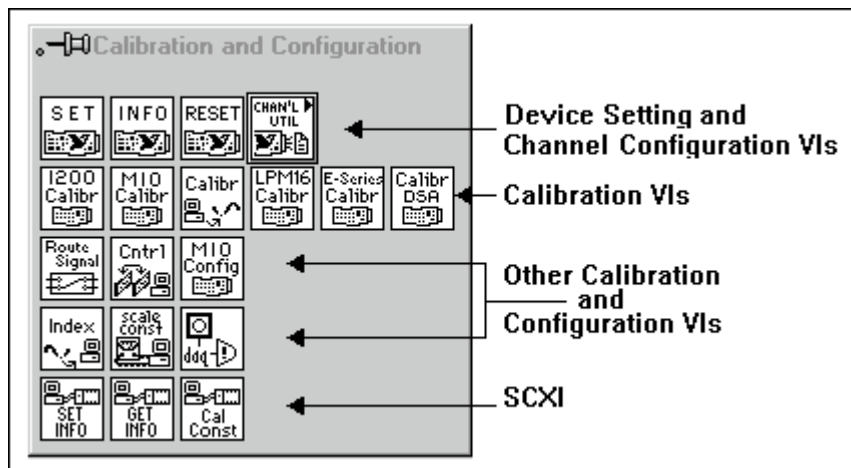
This chapter describes the VIs that calibrate specific devices and set and return configuration information.

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.



The following VIs only exist in the DAQ VI Library:

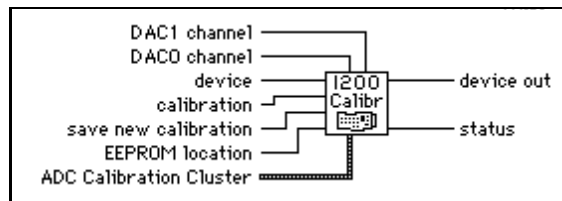
- A2000 Calibrate
- A2000 Configure
- A2100 Calibrate
- A2100 Config
- A2150 Calibrate
- A2150 Config
- DSP 2200 Calibrate
- DSP 2200 Configure

Calibration and Configuration VI Descriptions

The following Calibration and Configuration VIs are available.

1200 Calibrate

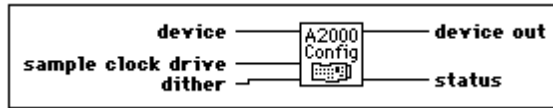
This VI calibrates the gain and offset values for the ADCs and DACs on 1200 Series devices (i.e., DAQPad-1200, DAQCard-1200, etc.).



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 reinstall NI-DAQ version 4.9.0 or an earlier version.*

Refer to Appendix B, [DAQ Hardware Capabilities](#), for more information on the NB-A2000 or EISA-A2000 DAQ devices.



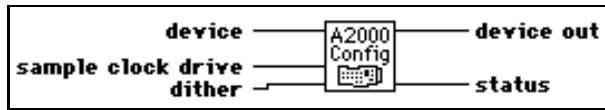
Warning *Read the calibration chapter in the NB-A2000 or EISA-A2000 User Manual before using the A2000 Calibrate VI.*

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 reinstall NI-DAQ version 4.9.0 or an earlier version.*

Refer to Appendix B, *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

Selects the desired calibration reference and performs an offset calibration cycle on the ADCs on the NB-A2100 or the NB-A2150.



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 reinstall NI-DAQ version 4.9.0 or an earlier version.*

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

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.

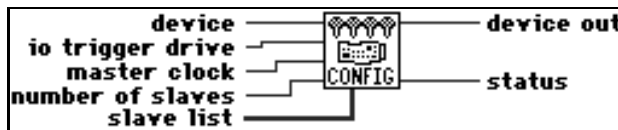


Warning *This VI is supported only up to NI-DAQ version 4.9.0 and has been removed from the Calibration and Configuration palette. This VI is still included in the DAQ VI Library for compatibility only, therefore if you are using NI-DAQ version 5.0 or later, this VI will return the following message: `deviceSupportError`. If you wish to use this VI, please reinstall NI-DAQ version 4.9.0 or an earlier version.*

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

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.



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 reinstall NI-DAQ version 4.9.0 or an earlier version.*

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.

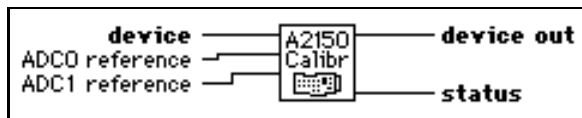
The NB-A2150 uses signals over the RTSI bus for sampling clock synchronization between two or more NB-A2150 devices. The sampling clock synchronization circuitry makes simultaneous sampling possible on more than four channels using additional NB-A2150 devices. If **master clock** is 1, **slave list** 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 (Macintosh)

Performs offset calibrations on the ADCs of the specified AT-A2150.



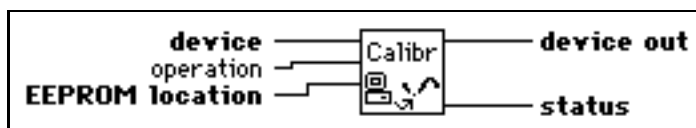
Warning *This VI is supported only up to NI-DAQ version 4.9.0 and has been removed from the Calibration and Configuration palette. This VI is still included in the DAQ VI Library for compatibility only, therefore if you are using NI-DAQ version 5.0 or later, this VI will return the following message: deviceSupportError. If you wish to use this VI, please reinstall NI-DAQ version 4.9.0 or an earlier version.*

Refer to Appendix B, [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 B, *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 B, *DAQ Hardware Capabilities*, for the channel limitations that apply to your device.

Table 29-1 shows possible values for the **channel scan list**, **channel list**, and **channel indices** parameters. Table 29-2 shows the possible values for the Sun. The **channel scan list** parameter is an input for the group configuration VIs.

Table 29-1. Channel to Index VI Parameter Examples

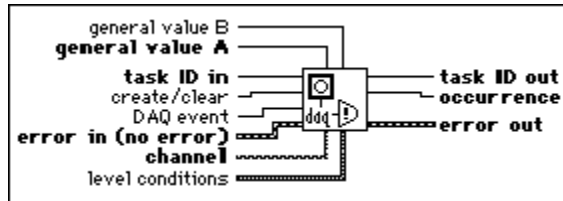
Channel Scan List	Channel List	Channel Indices
1, 3, 4, 5, 7	channel list [0] = 5	channel indices [0] = 3. Data for channel 5 is at position 3 within a scan. Indices are zero-based.
1, 3, 4, 5, 7	channel list is of 0 length.	channel indices is of 0 length. (In this case, status is non-zero.)
1, 2, 1, 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.
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 29-2. 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.

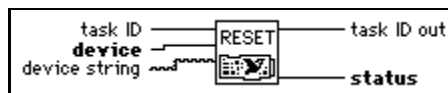
LabVIEW returns a Not a Refnum file I/O constant along with a non-zero status code if it cannot create the occurrence.

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

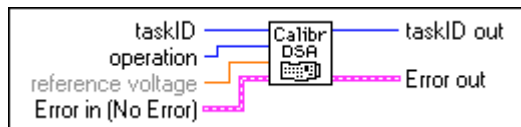
Resets either an entire device or the particular function identified by **taskID**.



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.

DSA Calibrate

Use this VI to calibrate your DSA device.

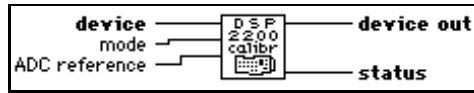


Your device contains calibration D/A converters (calDACs) that fine-tune the analog circuitry. The calDACs must be programmed (loaded) with certain numbers called calibration constants. These constants are stored in non-volatile memory (EEPROM) on your device. To achieve specification accuracy, perform an internal calibration of your device just before a measurement session but after your computer and the device have been running for at least 15 minutes. Frequent calibration produces the most stable and repeatable measurement performance.

Before the device is shipped from the factory, an external calibration is performed, and the EEPROM contains calibration constants that LabVIEW automatically loads into the calDACs as needed. The value of the onboard reference voltage is also stored in the EEPROM, and this value is used when you subsequently perform a self-calibration. The calibration constants are re-calculated and stored in the EEPROM when a self-calibration is performed. When you perform an external calibration, LabVIEW recalculates the value of the onboard reference voltage, and then performs a self-calibration. This new onboard reference value is used for all subsequent self-calibration operations. If a mistake is made when performing an external calibration, you can restore the board's factory calibration so that the board is not unusable.

DSP2200 Calibrate (Windows)

Performs offset calibrations on the analog input and/or analog output of the AT-DSP2200.



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 reinstall NI-DAQ version 4.9.0 or an earlier version.*

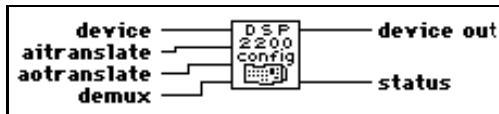
Refer to Appendix B, [DAQ Hardware Capabilities](#), for more information on the AT-DSP2200 DAQ device.

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.



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 reinstall NI-DAQ version 4.9.0 or an earlier version.*

Refer to Appendix B, [DAQ Hardware Capabilities](#), for more information on the AT-DSP2200 DAQ device.

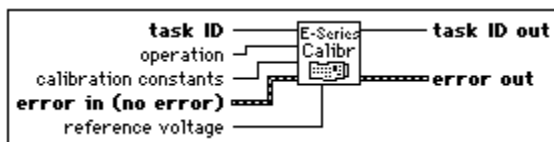
Because software running locally on the AT&T WE DSP32C DSP chip reads data from the ADCs and writes data to the DACs, you can manipulate the data during these transfers. When you write analog input data to DSP memory, you can write the data as unscaled 16-bit integers, unscaled 32C floating-point numbers, 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.

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

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

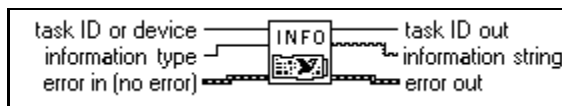
- Connect the positive output of your reference voltage source to the analog input channel 8.
- Connect the negative output of your reference voltage source to the AISENSE line.
- Connect DAC0 line (analog output channel 0) with analog input channel 0.
- If your reference voltage source and your computer 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

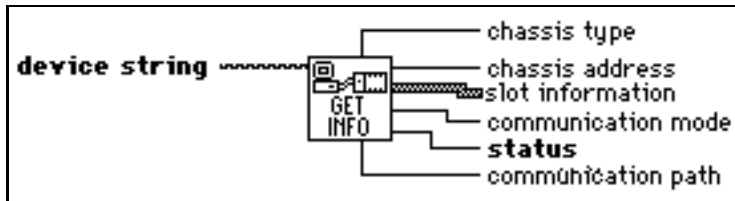
Returns information about a DAQ device.



Refer to Appendix B, *DAQ Hardware Capabilities*, for the transfer methods available with your DAQ device.

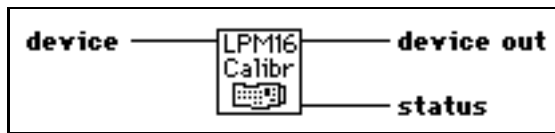
Get SCXI Information

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-16nP 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 B, *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 reinstall NI-DAQ version 4.9.0 or an earlier version.*

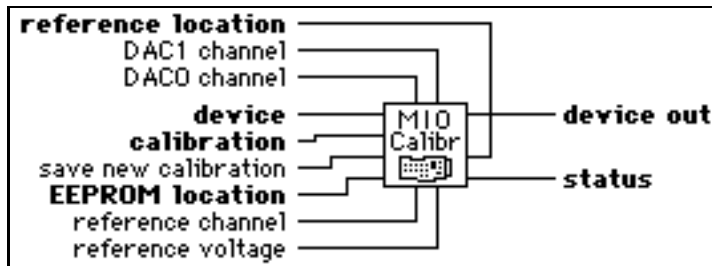
Makes sure LabVIEW always re-enables the slave devices *before* the master device in a multiple-buffer analog input operation. Only the following devices, which support multiple buffered acquisitions, can use this VI.

- (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 B, *DAQ Hardware Capabilities*, for more information on the AT-MIO-16F-5, AT-MIO-64F-5, and AT-MIO-16X DAQ devices.



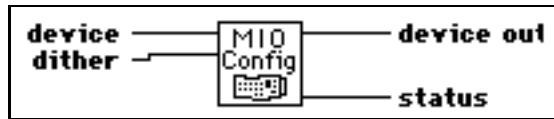
Note *You should always calibrate the ADC and the DACs after you calibrate the internal reference voltage.*



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.



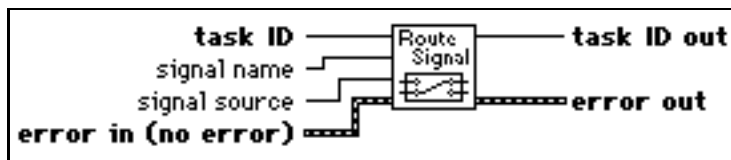
Refer to Appendix B, *DAQ Hardware Capabilities*, for more information on the devices supported by this VI.

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.

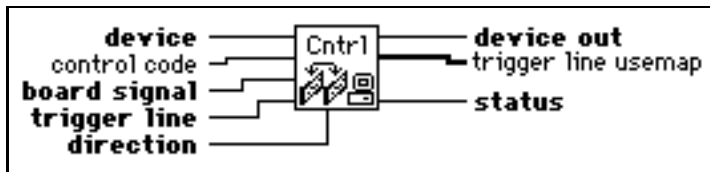


Note This VI is supported by E-Series and 54XX Series devices only.



RTSI Control

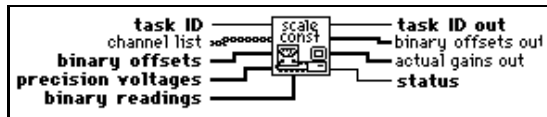
Connects or disconnects trigger and timing signals between DAQ devices along the Real-Time System Integration (RTSI) bus.



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.

Scaling Constant Tuner

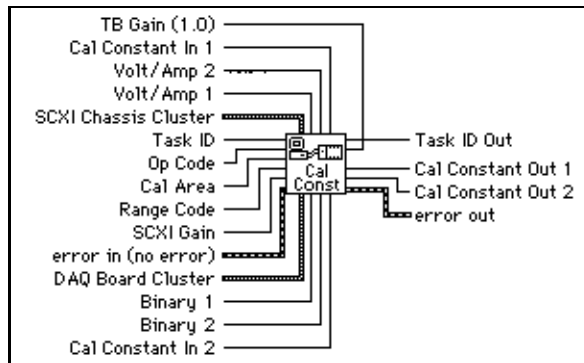
Adjusts the scaling constants, which LabVIEW uses to account for offset and non-ideal gain, to convert analog input binary data to voltage data.



For more information on the Scaling Constant Tuner VI, see the Scaling Constant Tuner VI description in Chapter 30, *Signal Conditioning VIs*.

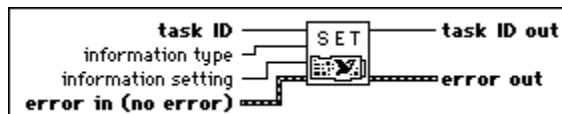
SCXI Cal Constants

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

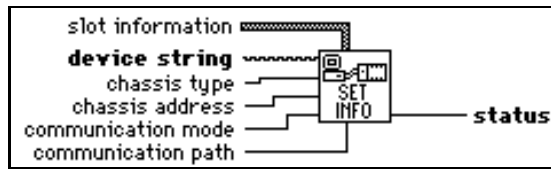
Sets the data transfer mode for different types of operations.



Refer to Appendix B, *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.



Get DAQ Channel Names

Returns an array of all the channel names in the default configuration file. A corresponding array of the channels' configured physical units is also returned. Using **channel type**, you can choose to retrieve all channels, or only analog input and analog output, or digital I/O channels.



Note

This VI is specific to computers running NI-DAQ 5.0 or later. LabVIEW returns an `UnsupportedError` message if you attempt to run this VI on computers not running NI-DAQ 5.0 or later.

Get Channel Information

Returns configuration information about a channel configured in the DAQ Channel Wizard.

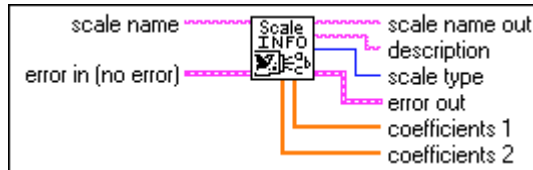


Note

This VI is specific to computers running NI-DAQ 5.0 or later. LabVIEW returns an `UnsupportedError` message if you attempt to run this VI on computers not running NI-DAQ 5.0 or later.

Get Scale Information

Returns configuration information about a scale configured in the DAQ Channel Wizard.



Note

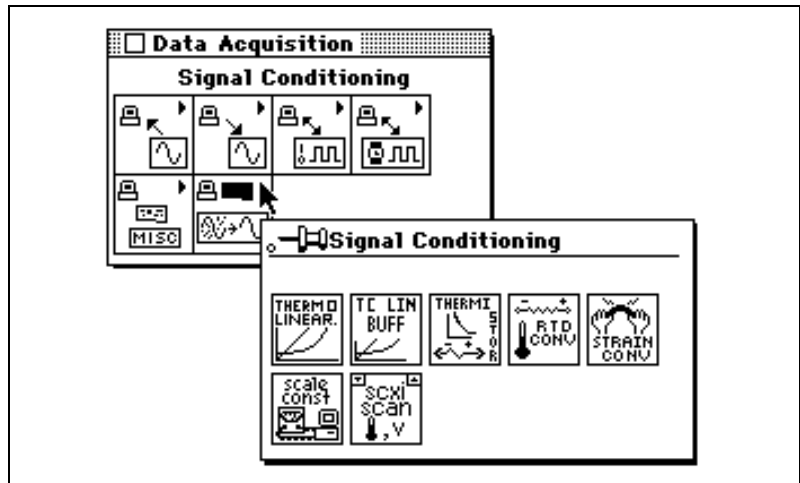
This VI is specific to computers running NI-DAQ 5.0 or later. LabVIEW returns an `UnsupportedError` message if you attempt to run this VI on computers not running NI-DAQ 5.0 or later.

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.

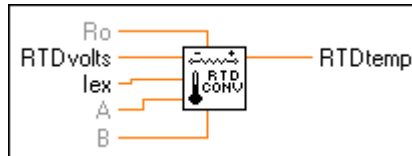


Signal Conditioning VI Descriptions

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:

$$R_t = R_0[1 + At + Bt^2 + C(t - 100)t^3]$$

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 **R₀** 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	A = 3.9784e-03
B = -5.80195e-07	B = -5.8408e-07
(α = 0.00385; ∂ = 1.492)	(α = 0.00392; ∂ = 1.492)

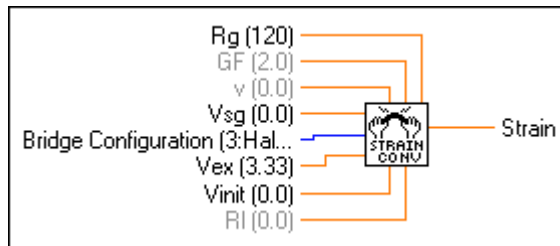
Some RTD documentation gives values for α and ∂ , from which you can calculate **A** and **B** using the following equations:

$$\mathbf{A} = \alpha(1 + \partial/100)$$

$$\mathbf{B} = -\alpha\partial/100^2$$

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 30-1 through 30-3 show the seven bridge configurations you can use and the corresponding formulas. For all bridge configurations, the VI uses the following formula to obtain V_r :

$$V_r = (V_{sg} - V_{init}) / V_{ex}$$

In the circuit diagrams, V_{OUT} is the voltage you measure and pass to the conversion VI as the V_{sg} parameter. In the quarter-bridge and half-bridge configurations, R_1 and R_2 are dummy resistors that are not directly incorporated into the conversion formula. The SCXI-1121 and SCXI-1122 modules provide R_1 and R_2 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 30-1 through 30-3 illustrate the bridge-completion networks available.

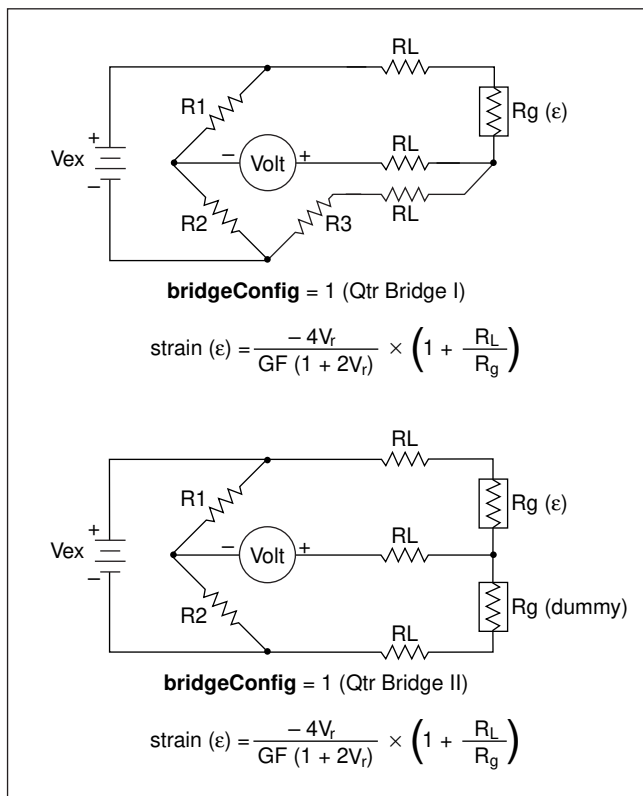


Figure 30-1. Strain Gauge Bridge Completion Networks (Quarter-Bridge Configuration)

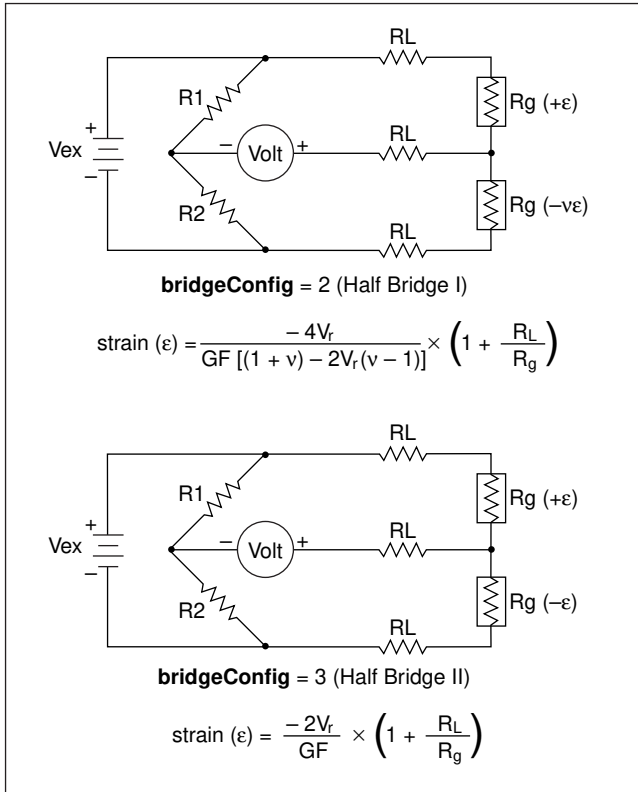


Figure 30-2. Strain Gauge Bridge Completion Networks (Half-Bridge Configuration)

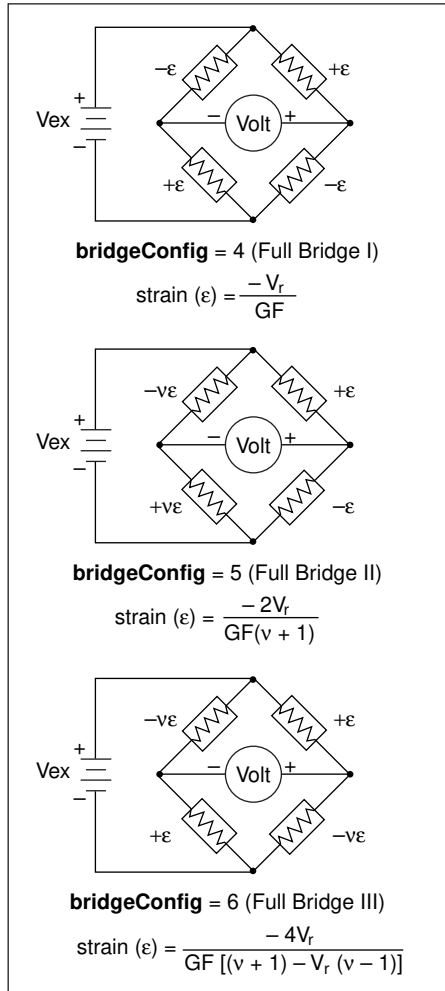
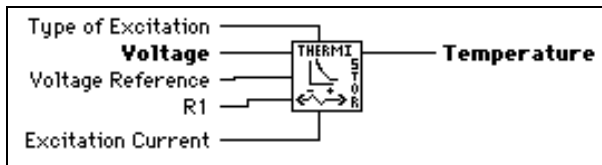


Figure 30-3. Strain Gauge Bridge Completion Networks (Full-Bridge Configuration)

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 30-4 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.

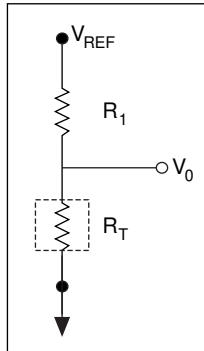


Figure 30-4. Circuit Diagram of a Thermistor in a Voltage Divider

Figure 30-5 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.

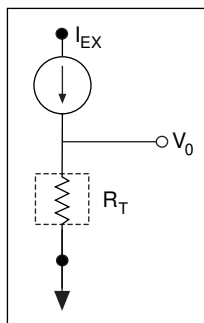


Figure 30-5. Circuit Diagram of a Thermistor with Current Excitation

If the thermistor is excited by voltage, the following shows equation relating the thermistor resistance, R_T , to the input values:

$$R_T = R_I \left(\frac{V_0}{V_{REF} - V_0} \right)$$

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:

$$T_K = \frac{1}{a + b(\ln R_T) + c(\ln R_T)^3}$$

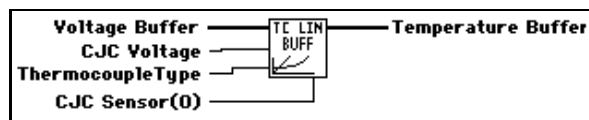
The values used by this VI for a , b , and c are given below. These values are correct for the thermistors provided on the SCXI and DAQPad-TB-52 terminal blocks. If you are using a thermistor with different values for a , b , and c (refer to your thermistor data sheet), you can edit the VI diagram to use your own a , b , and c values.

$$\begin{aligned} a &= 1.295361\text{E-}3 \\ b &= 2.343159\text{E-}4 \\ c &= 1.018703\text{E-}7 \end{aligned}$$

The VI produces a temperature in degrees Celsius. Therefore, $T_C = T_K - 273.15$.

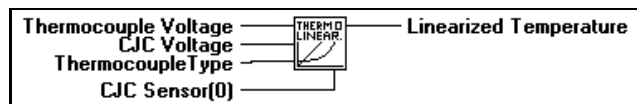
Convert Thermocouple Buffer

Converts a voltage buffer read from a thermocouple into a temperature buffer value in degrees Celsius.



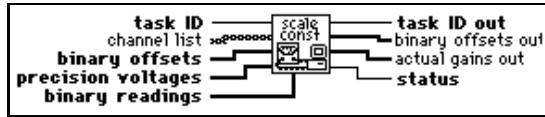
Convert Thermocouple Reading

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 21, *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** to this VI, the first element (at index 0) of the input arrays (**binary offsets**, **precision voltages**, and **binary readings**) apply 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.

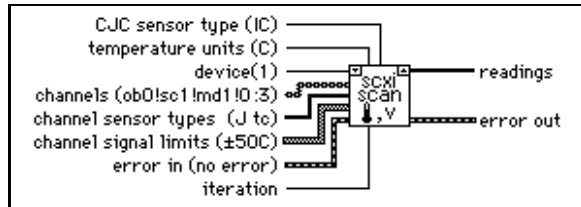
If you want to adjust only the channel offsets, and you want to assume the gain settings on the DAQ device and SCXI modules are ideal, wire only **binary offsets** and leave **precision voltages** and **binary readings** unwired.

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 channels. 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.



Instrument I/O Functions and VIs

Part III, *Instrument I/O Functions and VIs*, describes LabVIEW instrument drivers and GPIB, serial port, instrument driver template, and VISA VIs and functions. This part contains the following chapters:

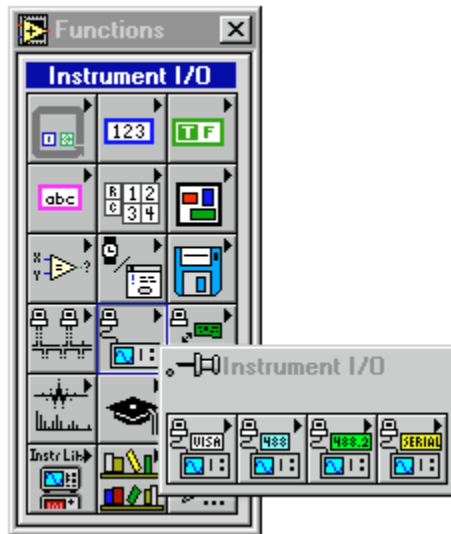
- Chapter 31, *Introduction to LabVIEW Instrument I/O VIs*, introduces LabVIEW instrument drivers and GPIB, serial port, instrument driver template, and VISA VIs and functions.
- Chapter 32, *Instrument Driver Template VIs*, describes the Instrument Driver Template VIs.
- Chapter 33, *VISA Library Reference*, describes the VISA Library Reference operations and attributes.
- Chapter 34, *Traditional GPIB Functions*, describes the traditional GPIB functions.
- Chapter 35, *GPIB 488.2 Functions*, describes the IEEE 488.2 (GPIB) functions.
- Chapter 36, *Serial Port VIs*, describes the VIs for serial port operations.

Introduction to LabVIEW Instrument I/O VIs

This chapter describes LabVIEW instrument drivers and GPIB, serial port, instrument driver template, and VISA VIs and functions.

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 also can 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 1, *Introduction to G Programming*, of the *G Programming Reference 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 about creating your own online reference files, see the *Creating Your Own Help Files* section in Chapter 5, *Printing and Documenting VIs*, of the *G Programming Reference 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 from National Instruments 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 a block diagram source code, so you can customize

them for your specific application, if needed. If a driver for your particular instrument does not exist, try one of the following suggestions:

- Use 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 and receive commands directly to and from your instrument.
- Refer to Chapter 7, *Getting Started with a LabVIEW Instrument Driver*, in the *LabVIEW User Manual* for information on how to start using LabVIEW instrument drivers from National Instruments.

Instrument Driver Distribution

LabVIEW instrument drivers are distributed in a variety of media including electronically via bulletin board and internet, and on 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 D, *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 32, *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 that 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 33, *VISA Library Reference*.

Introduction to GPIB

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

LabVIEW Traditional GPIB Functions

These traditional GPIB functions are compatible with both IEEE 488 and IEEE 488.2 devices and are sufficient for most applications. For more complex applications, such as using several devices and more than one GPIB interface, you can use the GPIB IEEE 488.2 functions.

For more information on the LabVIEW Traditional GPIB functions, see Chapter 34, *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 use 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 with 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 35, [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 35, [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 35, [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 35, [GPIB 488.2 Functions](#).

General Functions

The general functions are useful for special situations.

For more information on general functions, see Chapter 35, [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 36, [Serial Port VIs](#).

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 Utility Clean Up Initialize
- PREFIX Utility Default Instrument Setup

They also contain PREFIX VI Tree, a VI Example Tree.

Rather than developing your own VIs to accomplish these tasks, you can 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 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 are similar to other instrument drivers in the library. Therefore, you have a higher level of familiarity and understanding when you work with multiple instrument drivers.

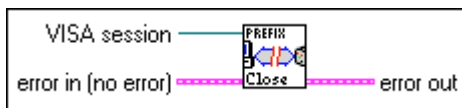
Instrument Driver Template VI Descriptions

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.*

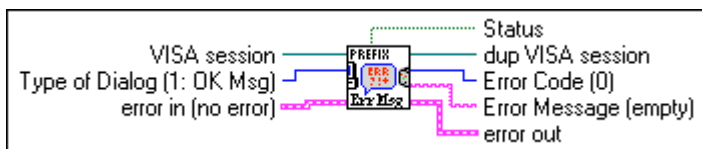
PREFIX Close

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.



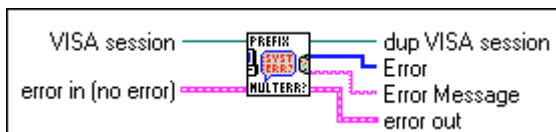
PREFIX Error Message

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.



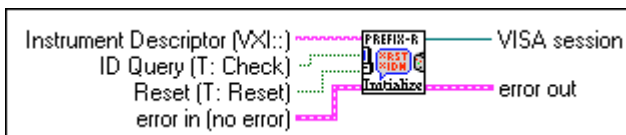
PREFIX Error Query, Error Query (Multiple) and Error Message

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]
Serial	ASRL[board] [::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 PREFIX 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.

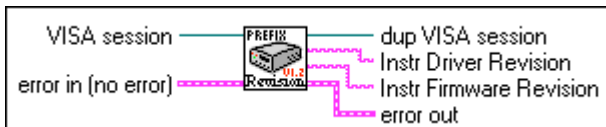
Also, you can call the Reset VI separately. If the instrument cannot perform reset, the Reset VI should return the literal string **Reset Not Supported**.



PREFIX Revision Query

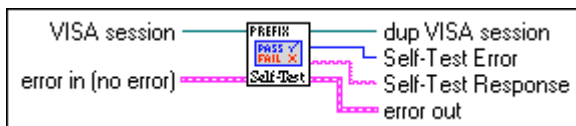
LabVIEW instrument drivers have a Revision Query VI. This VI outputs the following:

- 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 **Firmware Revision Not Supported**.)



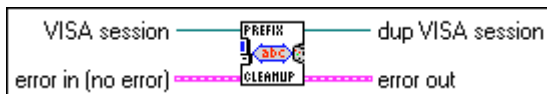
PREFIX Self-Test

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. If the instrument cannot perform a self-test, the Self-Test VI returns the literal string **Self-Test Not Supported**.



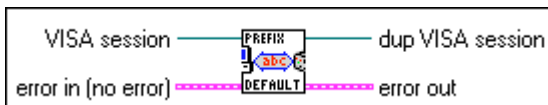
PREFIX Utility Clean UP Initialize

Closes an open VISA session if 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

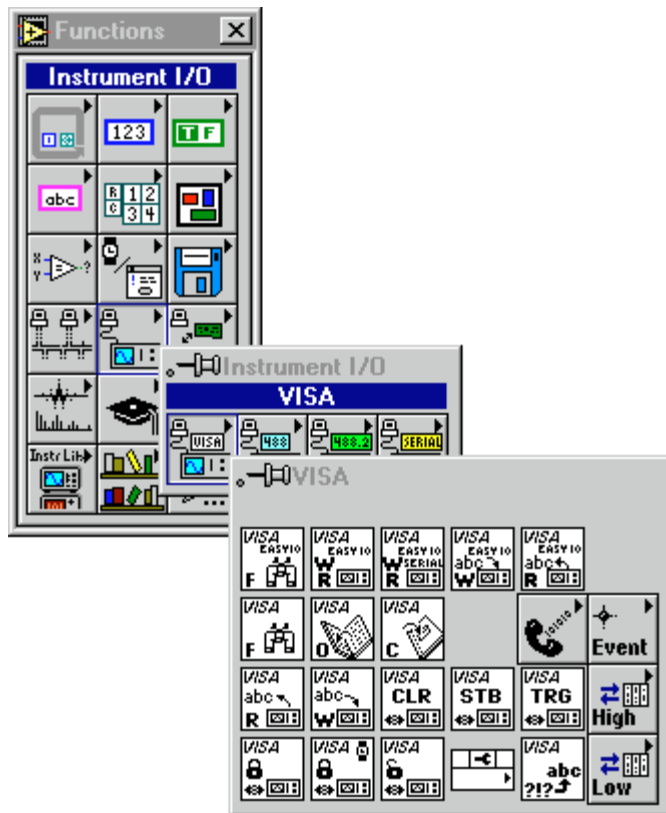
The VI Tree VI is a non-executable VI that shows the functional structure of the instrument driver. It contains the Getting Started VI, application VIs, and all of the component VIs.



VISA Library Reference

This chapter describes 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
- Serial Functions

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 used to communicate with a resource. It is created and linked to a resource by the VISA Open function. It then is used by other VISA functions to access the resource and its attributes. The dup VISA session is a copy of the VISA session that is passed out of the VISA functions. By passing the VISA session in and out of functions, you can simplify dataflow programming by chaining functions together. This is similar to the dup file refnums used by the File I/O functions.

VISA session is set to the **Instr** class by default. You can change the class type by popping up on the VISA session control in edit mode and selecting a different class. The following classes currently are supported:

- Instr
- GPIB Instr
- VXI/GPIB-VXI/VME 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

- PXI Instr
- VXI/GPIB-VXI/VME MemAcc

**Note**


The Generic Event, Service Request Event, Trigger Event, VXI Signal Event, VXI/VME Interrupt Event, and Resource Manager classes can only be passed in as a VISA session with the VISA Close function and the VISA Property Node.

VISA functions vary in the class of **VISA session** that can be wired to them. The valid classes for each function are indicated in the documentation. For example, the functions on the **High Level Register Access** and **Low Level Register Access** palettes do not accept VISA sessions of class GPIB Instr or Serial Instr. If you wire **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. An error cluster contains three fields. The status field is a Boolean that is TRUE when an error occurs and FALSE when no error occurs. **code field** is a VISA error code value if an error occurs during a VISA function. Appendix A, [Error Codes](#), lists the VISA Reference Library error codes. **source field** is a string that 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.

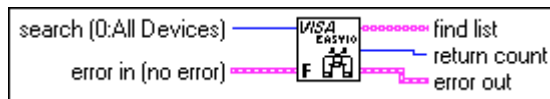
VISA Operation Descriptions

These functions appear on the main **VISA** palette. The valid classes for these functions are Instr (default), GPIB Instr, Serial Instr, VXI/GPIB-VXI/VME RBD Instr, and VXI/GPIB-VXI MBD Instr.

 **Note** *The following Easy VISA VIs provide a simple interface to the functions they use. If optimizing performance is important for your application, use the VISA primitives, also located in this palette.*

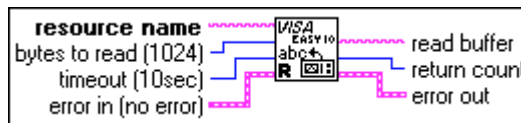
Easy VISA Find Resources

Finds all the VXI, Serial, and GPIB resources that are available for communication.



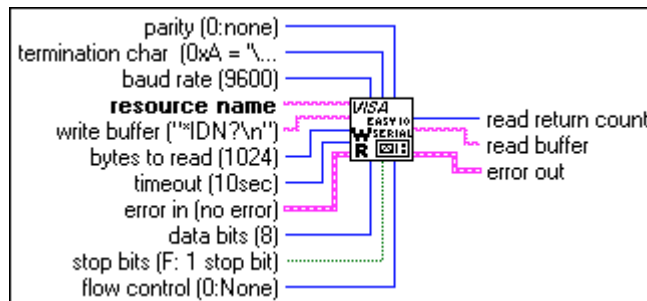
Easy VISA Read

Reads data from the resource specified by the resource name. The maximum number of bytes to be read is determined by the byte count.



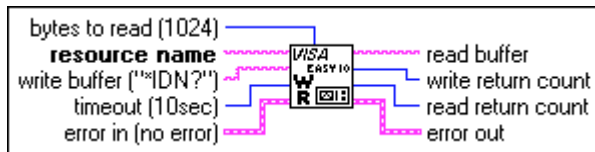
Easy VISA Serial Write and Read

Writes a command string to the specified serial device then reads the response data. The read is terminated when the specified termination character is received or after receiving the number of bytes specified in the bytes to read parameter, whichever is first. If a termination character is required for writing to the instrument, it needs to be included in the write buffer string.



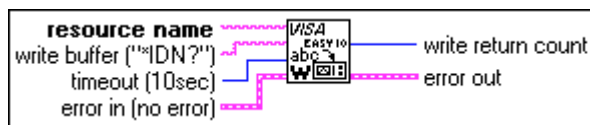
Easy VISA Write and Read

Writes a command string to the specified device then reads the response data.



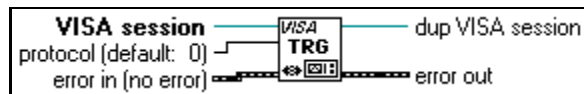
Easy VISA Write

Writes a command string to the specified device.



VISA Assert Trigger

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



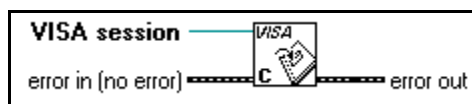
VISA Clear

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. For Serial, this sends the string *CLS In.



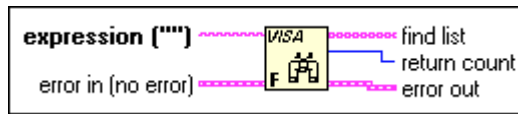
VISA Close

Closes a specified device session or event object. VISA Close accepts all available classes. For a listing of available classes, see the *VISA Library Reference Parameters* section earlier in this chapter.



VISA Find Resource

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



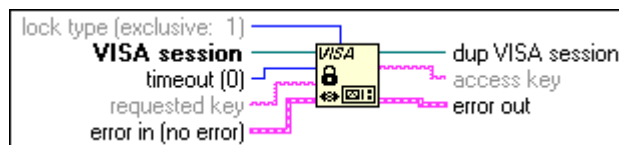
The following tables show the expression parameter descriptions for the VISA Find Resource VI.

Instrument Resources	Expression
GPIB	GPIB[0-9]*: :?*INSTR
VXI	VXI?*INSTR
GPIB-VXI	GPIB-VXI?*INSTR
GPIB and GPIB-VXI	GPIB?*INSTR
All VXI	?*VXI[0-9]*: :?*INSTR
Serial	ASRL[0-9]*: :?*INSTR
All	?*INSTR

Memory Resources	Expression
VXI	VXI?*MEMACC
GPIB-VXI	GPIB-VXI?*MEMACC
All VXI	?*VXI[0-9]*: :?*MEMACC
All	?*MEMACC

VISA Lock

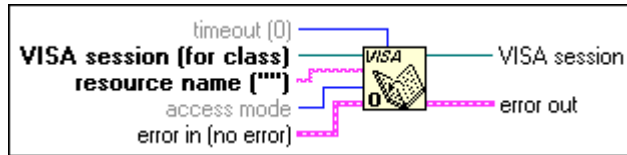
Establishes locked access to the specified resource.



For more information about VISA locking and shared locking, refer to Chapter 8, *LabVIEW VISA Tutorial*, in the *LabVIEW User Manual*.

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 string segments are shown in square brackets ([]).

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

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

The following table shows the default value for optional string segments.

Optional String Segments	Default Value
board	0
secondary address	none

The following table shows examples of address strings.

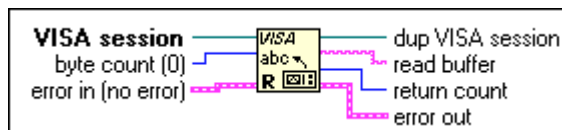
Address String	Description
VXI0::1::INSTR	A VXI device at logical address 1 in VXI interface VXI0.
GPIB-VXI::9::INSTR	A VXI device at logical address 9 in a GPIB-VXI controlled system.
GPIB::1::0::INSTR	A GPIB device at primary address 1, secondary address 0 in GPIB interface 0.
ASRL1::INSTR	A serial device attached to interface ASRL1.
VXI::MEMACC	Board-level register access to the VXI interface.
GPIB-VXI1::MEMACC	Board-level register access to GPIB-VXI interface number 1.

For the access mode parameter, the value `VI_EXCLUSIVE_LOCK` (1) is used to acquire an exclusive lock immediately upon opening a session; if a lock cannot be acquired, the session is closed and an error is returned.

The value `VI_LOAD_CONFIG` (4) is used to configure attributes to values specified by some external configuration utility, such as T&M Explorer (on Windows 95/NT) or VISAconf (on Windows 3.x, Solaris 2, and HP-UX).

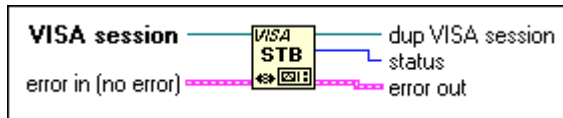
VISA Read

Reads data from a device. Whether the data is transferred synchronously or asynchronously is platform-dependent.



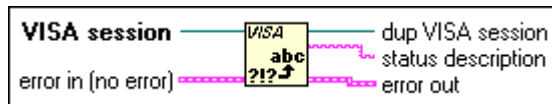
VISA Read STB

Reads the 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.



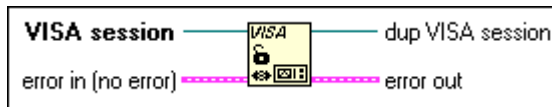
VISA Status Description

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



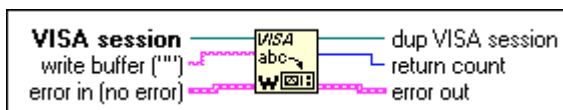
VISA Unlock

Relinquishes the lock previously obtained using the VISA Lock function.



VISA Write

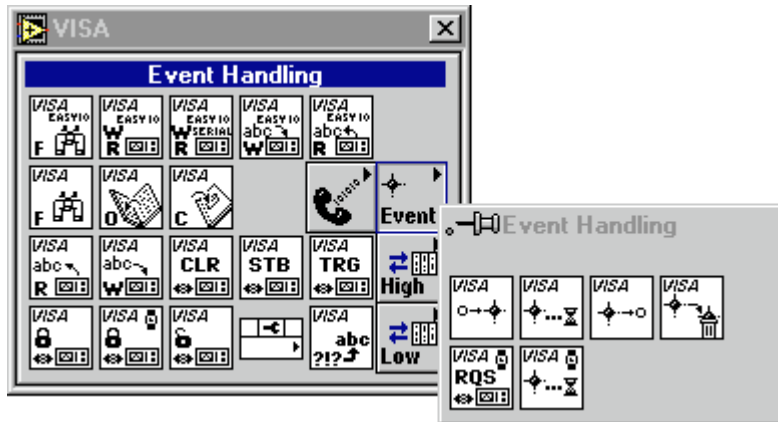
Writes data to the device. Whether the data is transferred synchronously or asynchronously is platform-dependent.



Event Handling Functions

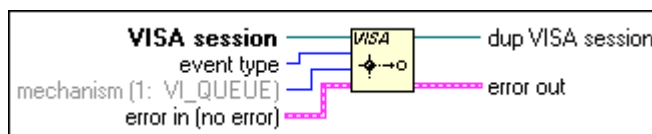
This section describes the VISA Event Handling functions. Valid classes for these functions are Instr (default), GPIB Instr, Serial Instr, VXI/GPIB-VXI/VME RBD Instr, and VXI/GPIB-VXI MBD Instr.

You can find the VISA Event Handling functions in the **VISA** palette, which you access by selecting **Functions»Instrument I/O»VISA**.



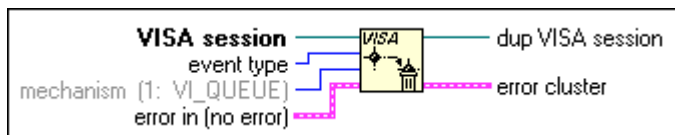
VISA Disable Event

Disables servicing of an event. This operation prevents new event occurrences from being queued. However, event occurrences already queued are not lost; use the VISA Discard Events VI if you want to discard queued events.



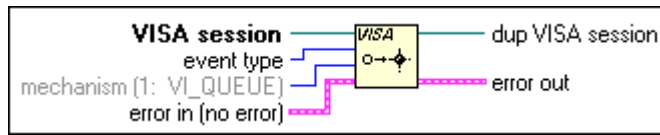
VISA Discard Events

Discards all pending occurrences of the specified event types and mechanisms from the specified session.



VISA Enable Event

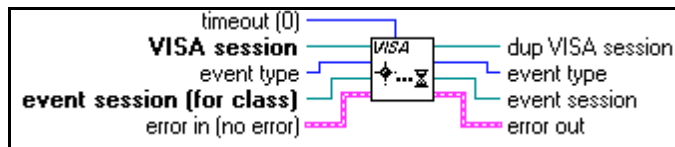
Enables notification of a specified event.



Note *You must call the VISA Enable Event VI for a given session before using VISA Wait on Event VI.*

VISA Wait On Event

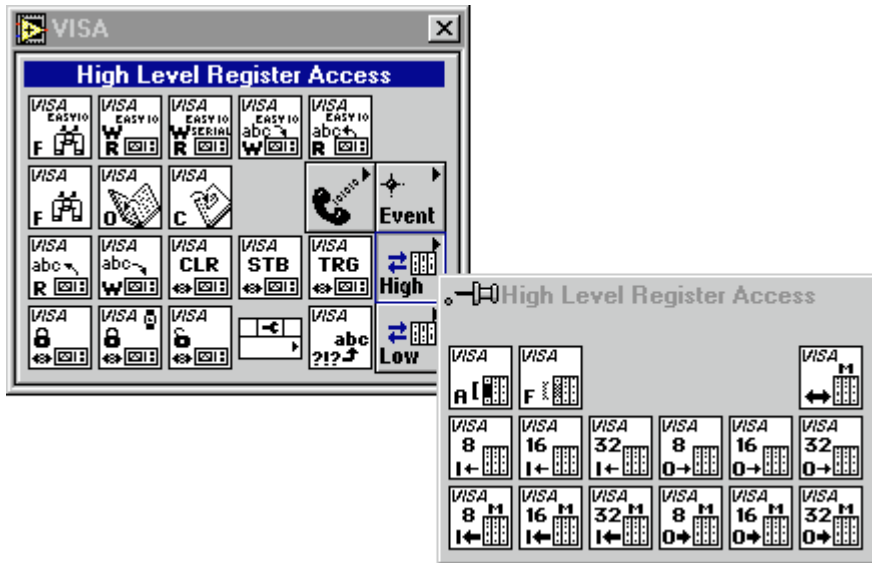
Suspends execution of a thread of application and waits for an event of **event type** for a time period not to exceed that specified by **timeout**. Refer to individual event descriptions for context definitions. If the specified **event type** is All Events, the operation waits for any event that is enabled for the given session.



Note *You must first call the VISA Enable Event VI for the specified session before using VISA Wait on Event VI.*

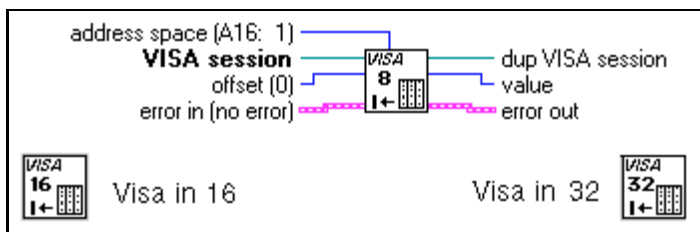
High Level Register Access Functions

This section describes the VISA High Level Register Access functions. Valid classes for these functions are Instr (default), VXI/GPIB-VXI/VME RBD Instr, VXI/GPIB-VXI MBD Instr, and VXI/GPIB-VXI/VME MemAcc. 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 8 bits, 16 bits, or 32 bits of data, respectively, from the specified memory space (assigned memory base plus offset). These functions do not require VISA Map Address to be called prior to their invocation.



The following table lists the valid entries for specifying address space.

Address Space Value	Description
(1)	Address the A16 address space of the VXI/MXI bus.
(2)	Address the A24 address space of the VXI/MXI bus.
(3)	Address the A32 address space of the VXI/MXI bus.

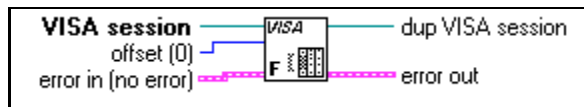
VISA Memory Allocation

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. If the device to which the given VISA Session refers is located on the local interface card, the memory can be allocated either on the device itself or on the computer's system memory. The memory region referenced by **offset** that is returned from this function can be accessed with the high-level functions VISA Move In8 / Move In16 / Move In32 and VISA Move Out8 / Move Out16 / Move Out32, or it can be mapped using the VISA MapAddress function.



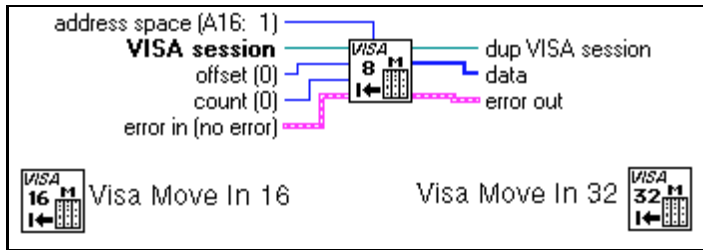
VISA Memory Free

Frees the memory previously allocated by the VISA Memory Allocation function. If the specified **offset** has been mapped using the VISA Map Address function, it must be unmapped before the memory can be freed.



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. The VISA Move InXX functions use the specified address space to read 8 bits, 16 bits, or 32 bits of data, respectively, from the specified offset. These functions do not require the VISA Map Address to be called prior to their invocation.

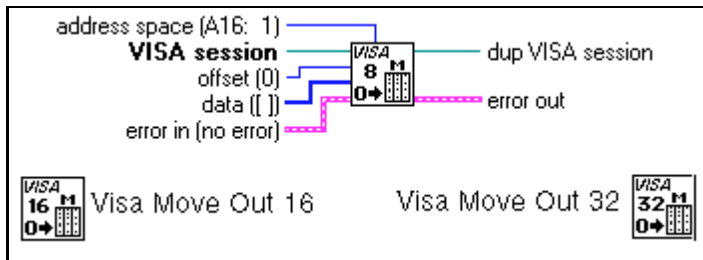


The following table lists the valid entries for specifying address space.

Value	Description
(1)	Address the A16 address space of the VXI/MXI bus.
(2)	Address the A24 address space of the VXI/MXI bus.
(3)	Address the A32 address space of the VXI/MXI bus.

VISA Move Out8 / Move Out16 / Move Out32

Moves a block of data from local memory to the specified address and offset and uses the specified address space to write 8 bits, 16 bits, or 32 bits of data, respectively, to the specified offset. These functions do not require the VISA Map Address function to be called prior to their invocation.

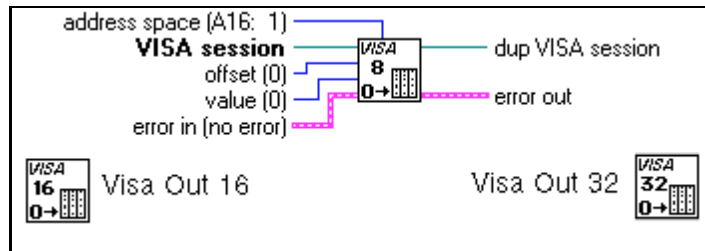


The following table lists the valid entries for specifying address space.

Value	Description
(1)	Address the A16 address space of the VXI/MXI bus.
(2)	Address the A24 address space of the VXI/MXI bus.
(3)	Address the A32 address space of the VXI/MXI bus.

VISA Out8 / Out16 / Out32

Writes 8 bits, 16 bits, or 32 bits of data, respectively, to the specified memory space (assigned memory base plus offset). These functions do not require the VISA Map Address function to be called prior to their invocation.

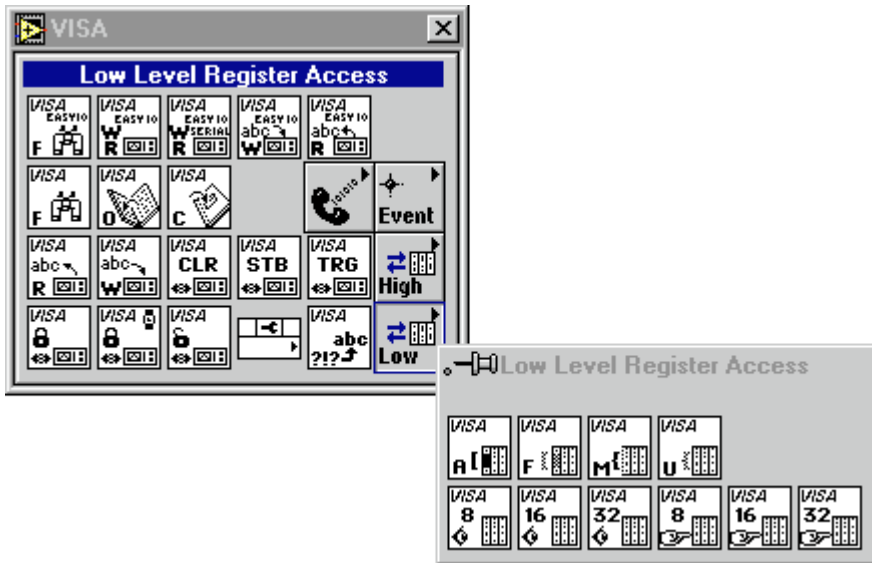


The following table lists the valid entries for specifying address space.

Value	Description
(1)	Address the A16 address space of the VXI/MXI bus.
(2)	Address the A24 address space of the VXI/MXI bus.
(3)	Address the A32 address space of the VXI/MXI bus.

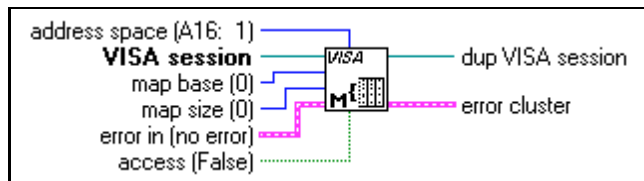
Low Level Register Access Functions

This section describes the VISA Low Level Register Access functions. Valid classes for these functions are Instr (default), VXI/GPIB-VXI VME MemAcc, VXI/GPIB-VXI RBD Instr, VXI/GPIB-VXI MBD Instr, and VXI/GPIB-VXI VME MemAcc. To access the VISA Low Level Register Access functions, pop up on the Low Level icon on the **VISA** palette.



VISA Map Address

Maps a specified memory space. The memory space that is mapped is dependent on the type of interface specified by **VISA session** and the **address space** parameter. Once the window is mapped, VISA tracks the window that is mapped. This behavior dictates that VISA can only map one window for each VISA session.



The following table lists the valid entries for specifying address space.

Value	Description
(1)	Address the A16 address space of the VXI/MXI bus.
(2)	Address the A24 address space of the VXI/MXI bus.
(3)	Address the A32 address space of the VXI/MXI bus.

VISA Memory Allocation

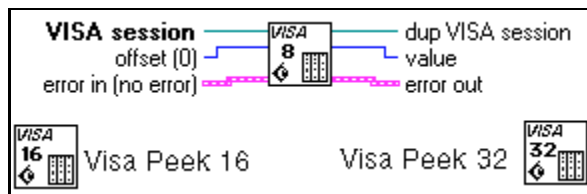
For information about the VISA Memory Allocation function, see the *High Level Register Access Functions* section of this chapter.

VISA Memory Free

For information about the VISA Memory Free function, see the *High Level Register Access Functions* section of this chapter.

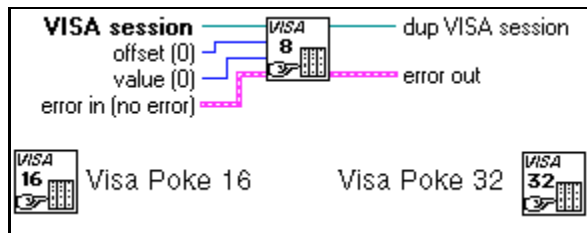
VISA Peek8 / Peek16 / Peek32

Reads an 8-bit, 16-bit, or 32-bit value, respectively, from the address location specified in **offset**. The address must be a valid memory address in the current process mapped by a previous VISA Map Address function call.



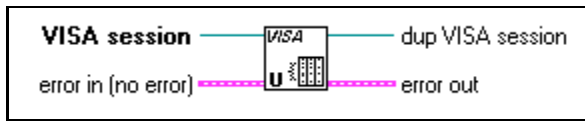
VISA Poke8 / Poke16 / Poke32

Writes an 8-bit, 16-bit, or 32-bit value, respectively, to the specified address and stores the content of the value to the address pointed to by **offset**. The address must be a valid memory address in the current process mapped by a previous VISA Map Address function call.



VISA Unmap Address

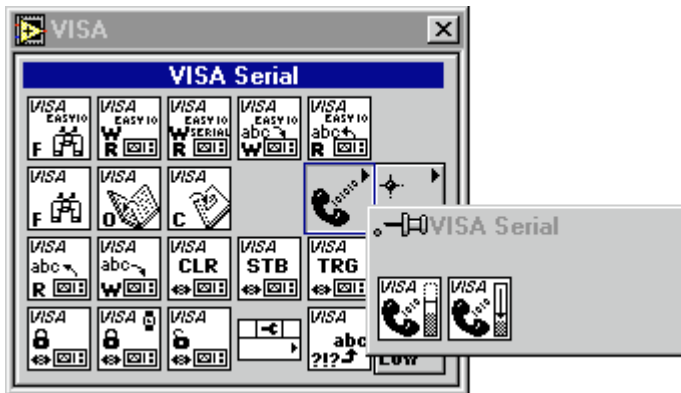
Unmaps memory space previously mapped by VISA Map Address.



VISA Serial Functions

This section describes the VISA functions that are specific to serial ports. Valid classes for these functions are Instr (default) and Serial Instr.

To access the VISA Serial functions, pop up on the Low Level icon on the VISA palette.



Flush Serial Buffer

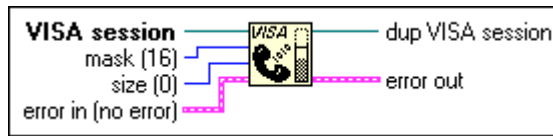
Flushes the serial buffer.



Flushing the receive buffer (16) discards the contents, while flushing the output buffer (32) waits for any remaining contents in the transmit buffer to be sent to the device. To discard any remaining data in the transmit buffer, you need to use the discard output buffer mask (128). To flush more than one buffer simultaneously, combine the buffer masks by using an O-Ring.

Set Serial Buffer Size

Sets the size of the serial buffer.

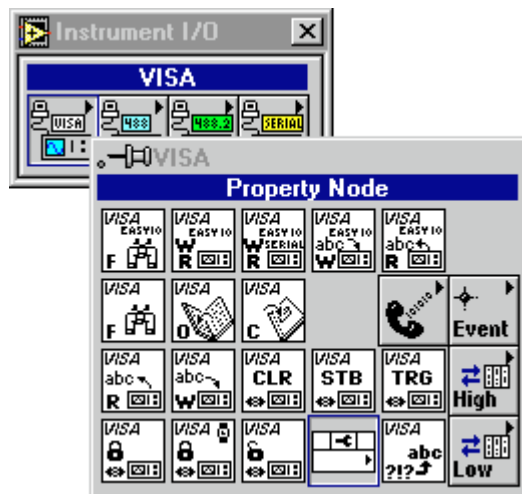


Valid values for **mask** are Serial receive buffer (16) and Serial transmit buffer (32). To set the size of both buffers simultaneously, combine the buffer masks by using an O-Ring.

VISA Property Node

This section describes the VISA Library attributes. The VISA Property Node gets and/or sets the indicated attributes. The node is expandable; evaluation starts from the top and proceeds downward until an error or until the final evaluation occurs.

To access the property node, select **Functions»Instrument I/O»VISA**. Then select the Property Node icon located on the bottom row of the **VISA** palette.



The VISA Property Node only displays attributes for the class of the session that is wired to it. You can change the class of a VISA Property Node as long as you have not wired it to a **VISA session**. Once a **VISA session** is wired to a VISA Property Node, it adapts to the class of

the session and any displayed attributes that are not valid for that class become invalid (indicated by turning the attribute item black).

Because the Property Node can be used in other contexts in LabVIEW, the Property Node might default to a class other than a VISA class if you place it on a diagram by itself. When you wire it to a **VISA session**, it becomes a VISA class.

VISA Property Node Descriptions

The following VISA Property Node categories are available.

Fast Data Channel

Specifies the following information:

- Channel number
- Data transfers through channel pairs
- Enabling signal
- Normal-mode or streaming-mode transfers

General Settings

Determine the following properties:

- Maximum event queue length
- Unique VISA resource name
- Resource lock state
- Timeout value for accessing the device
- Communication trigger mechanism
- Information used to document the functionality in your VISA application

GPIO Settings

Specify the following information:

- What the primary and secondary addresses of a GPIO device are
- If the GPIO device needs to be readdressed before every transfer
- If the GPIO device is unaddressed after each read and write operation

Interface Information

Provides information about the VISA interface type, the board number of the interface and the board number of the parent device.

Message-Based Settings

Determine the following aspects of VISA message-based communication:

- The I/O Protocol (GPIB, Serial, VXI)
- Whether to send an END indicator in write operations
- Whether to ignore an END indicator in read operations
- Whether to terminate read operations with a special character

Modem Line Settings

Determine the current state of the following signals used in modem communication: CTS, DCD, DSR, DTR, RI, and RTS.

PXI Resources

Specify the address type, the address base and address size of devices at slots BAR0 through BAR5.

PXI Settings

Specify the following information: device number, function number, subsystem manufacturer information, and subsystem model code.

Register-Based Settings

Determine the following aspects of VISA register-based communication:

- Identification of the device manufacturer
- Model of the device
- Physical slot location of the device
- Number of elements in block-move operations at both the source and memory addresses
- **(Windows)** Base address, size, and access to space

Serial Settings

Specify the following: number of bytes at the serial port, baud rate, data bits, parity, stop bits, flow control, and termination method for read and write operations.

Version Information

Provides information about the version and the manufacturer's name of the VISA implementation, as well as the version of the VISA specification.

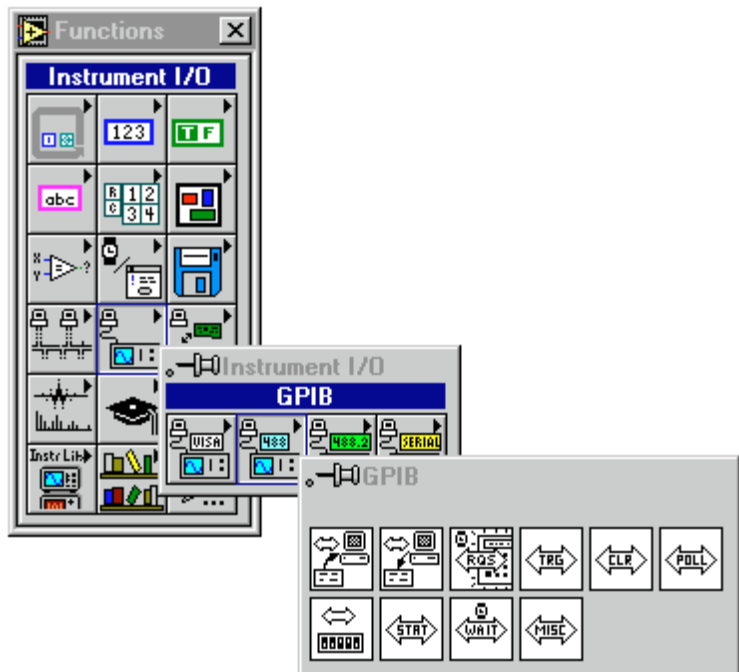
VME/VXE Settings

Determine the necessary addresses, access privileges, memory space, and byte orders necessary for VXI communication.

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\smp1gpib.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, do not specify **address string**.

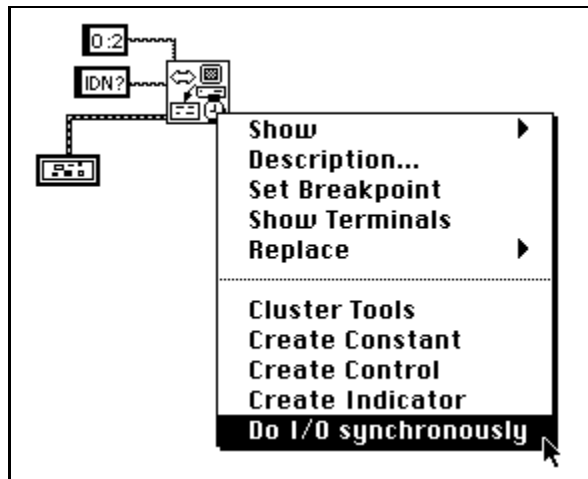
When there are multiple GPIB Controllers that LabVIEW can use, a prefix to **address string** in the form `ID:address` (or `ID:` if no address is necessary) determines the Controller that a specific function uses. If a Controller ID is not present, the functions assume Controller (or bus) 0.

- **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.
- **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. **code field** will be a GPIB error code value if an error occurs during a GPIB function. **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 functioning 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 operate. When set to execute asynchronously, a small wristwatch icon appears as part of the function icons. A pop-up item on the Traditional GPIB Read and GPIB Write functions allows you to switch their behavior to and from asynchronous operation.

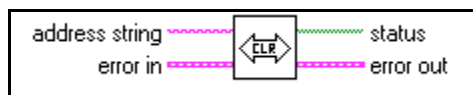


Traditional GPIB Function Descriptions

The following traditional GPIB functions are available.

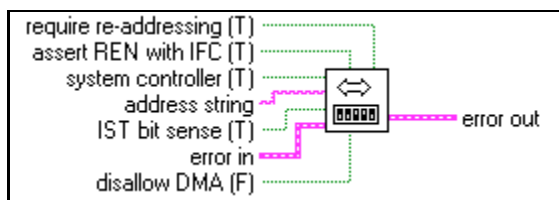
GPIB Clear

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



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.

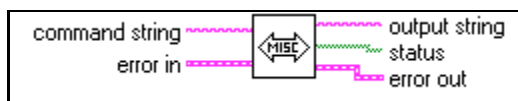


Table 34-1. Command String Device Functions

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).

Table 34-2. Command String Controller Functions

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.
llo	Local lockout.
loc	Place Controller in local state.

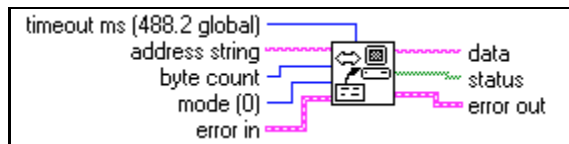
Table 34-2. Command String Controller Functions (Continued)

Controller Functions	Description
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 and set remote enable.
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

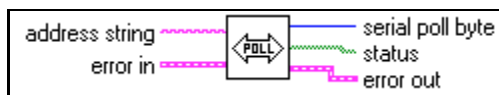
Reads **byte count** number of bytes from the GPIB device at **address string**.



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 35, [GPIB 488.2 Functions](#), for more information.

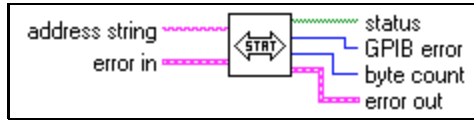
GPIB Serial Poll

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



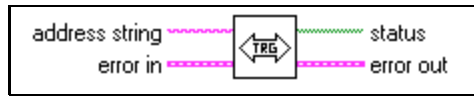
GPIB Status

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



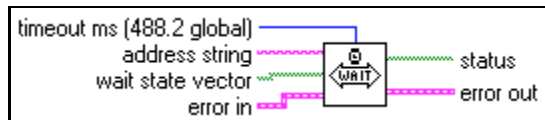
GPIB Trigger

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



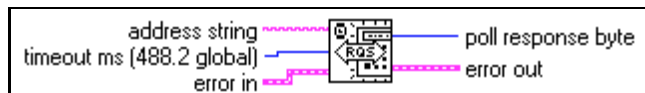
GPIB Wait

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



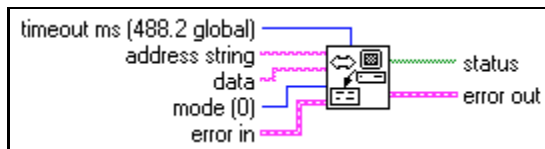
Wait for GPIB RQS

Waits for the device indicated by **address string** to assert RQS.



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 Go To Local (GTL) 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 becomes idle automatically. 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. Taking control synchronously 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 RSQ line. For instance, if you want to assert the GPIB RSQ line, send the ASCII character @, in which the 0x40 bit is set.

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

sic – Send interface clear

syntax **sic**

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

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

sre – Unassert or assert remote enable

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

sre 1 (assert Remote Enable)

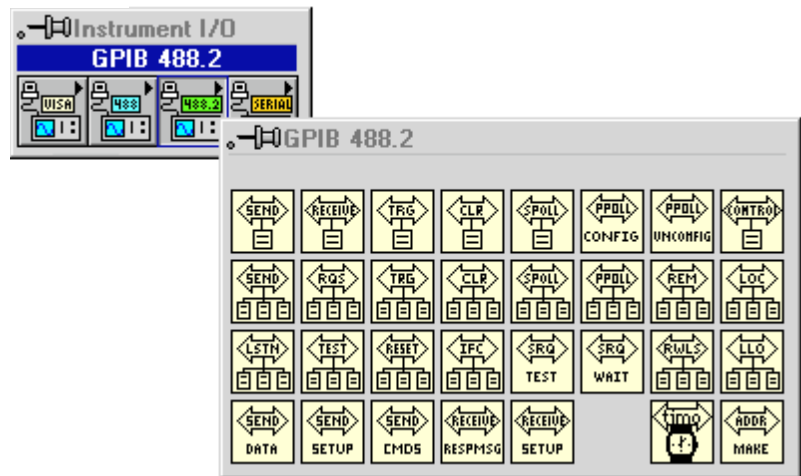
sre unasserts or asserts the GPIB REN line. Devices monitor REN when they select between local and remote modes of operation. A device does not actually enter remote mode until it receives its listen address.

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

GPIB 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.

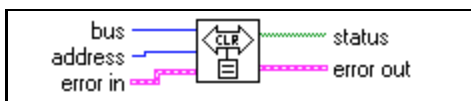
- **bus** refers to the GPIB bus number. If you have only one GPIB interface in your computer, the default bus number is 0. For additional GPIB interfaces, see the software installation instructions included with your GPIB board.
- **byte count** refers to the number of bytes that pass over the GPIB.
- **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.
- **error in; error out.** See the *GPIB Traditional Function Parameters* section of Chapter 34, *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.

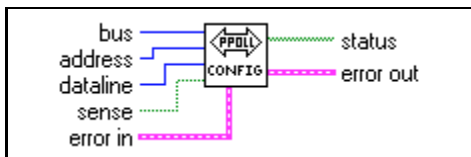
DevClear

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



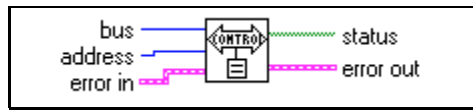
PPollConfig

Configures a device for parallel polls.



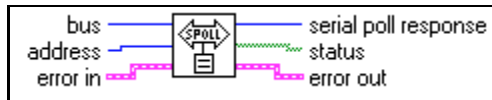
PassControl

Passes control to another device with Controller capability.



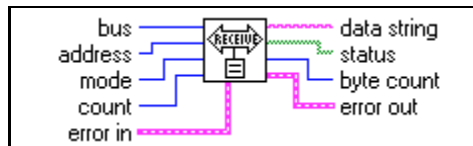
ReadStatus

Serial polls a single device to get its status byte.



Receive

Reads data bytes from a GPIB device.

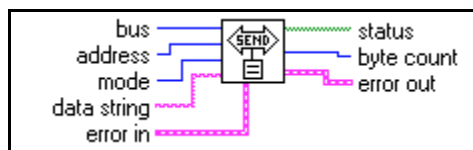


Receive terminates when the function does one of the following:

- 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)

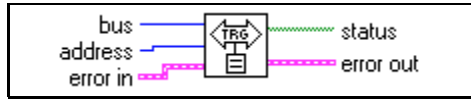
Send

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.

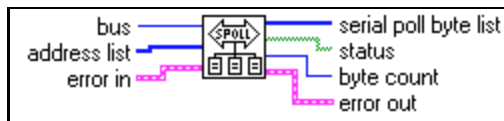


GPIB 488.2 Multiple-Device Function Descriptions

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

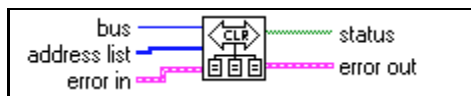
Serial polls all devices.



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.

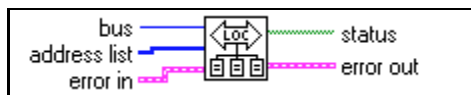
DevClearList

Clears multiple devices simultaneously.



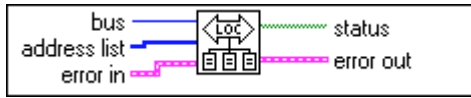
EnableLocal

Enables local mode for multiple devices.



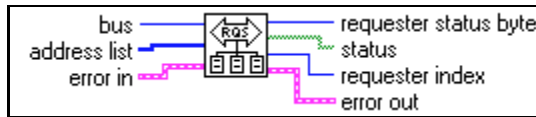
EnableRemote

Enables remote programming of multiple GPIB devices.



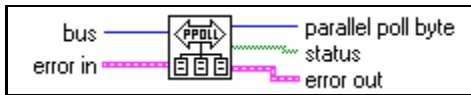
FindRQS

Determines which device is requesting service.



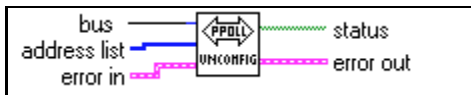
PPoll

Performs a parallel poll.



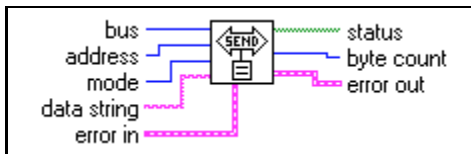
PPollUnconfig

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.



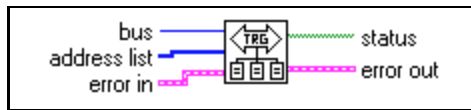
SendList

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.

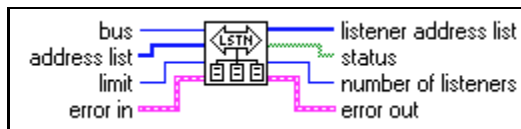


GPIB 488.2 Bus Management Function Descriptions

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 to determine their identity.

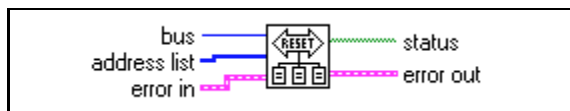


ResetSys

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.

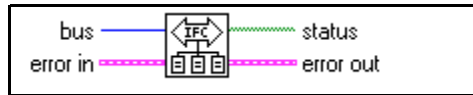
Third, the function sends the *RST message to all devices whose addresses are contained in the **address list** array. This message initializes device-specific functions within each device.



SendIFC

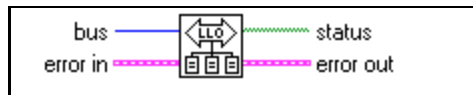
Clears the GPIB functions with Interface Clear (IFC). When you issue the GPIB Device IFC message, the interface functions of all connected devices return to their cleared states.

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.



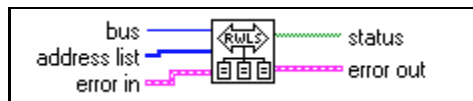
SendLLO

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.



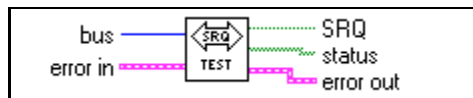
SetRWLS

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.



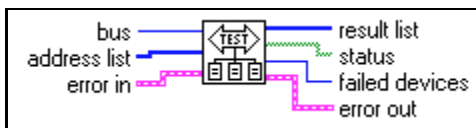
TestSRQ

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

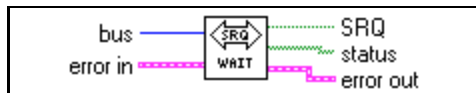
Directs multiple devices to conduct IEEE 488.2 self-tests.



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.

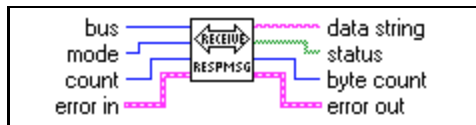


GPIB 488.2 Low-Level I/O Function Descriptions

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.



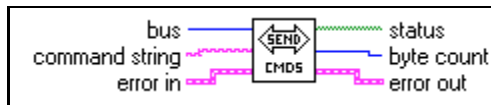
ReceiveSetup

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 readdress the devices between blocks of reads.



SendCmds

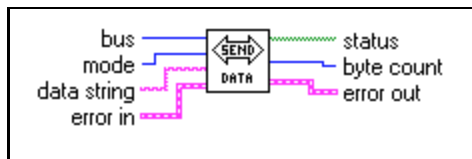
Sends GPIB command bytes.



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.

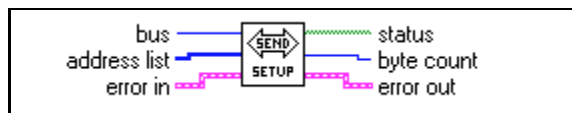
SendDataBytes

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 readdress the devices between blocks of sends.

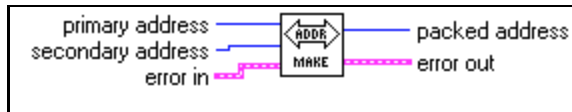


GPIB 488.2 General Function Descriptions

The general functions are useful for special situations.

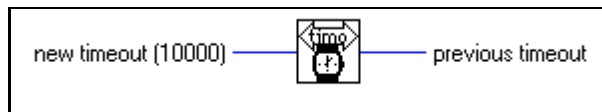
MakeAddr

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



SetTimeOut

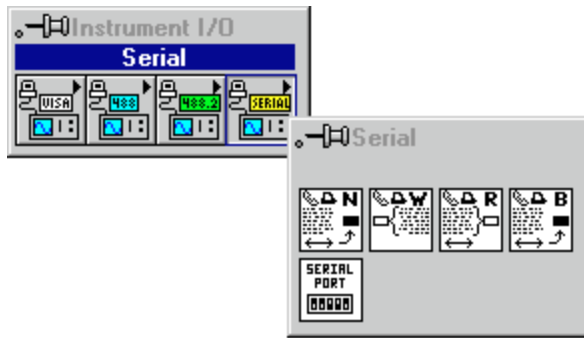
Changes the global timeout period for all GPIB 488.2 functions. This function also sets the default timeout period for all GPIB functions.



Serial Port VIs

This chapter describes the VIs for serial port operations.

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



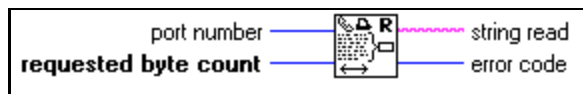
For examples of how to use the Serial Port VIs, see `examples\instr\smp1ser1.llb`.

Serial Port VI Descriptions

The following Serial Port VIs are available.

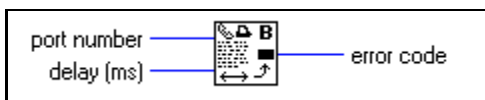
Bytes at Serial Port

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



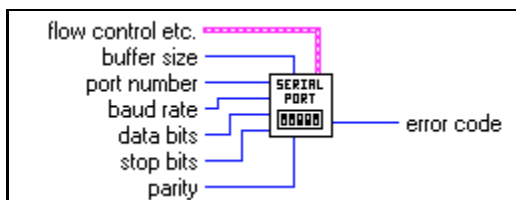
Serial Port Break

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.



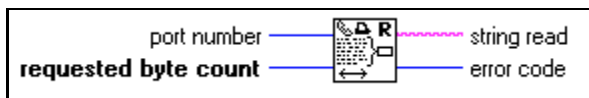
Serial Port Init

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**.



Analysis VIs

Part IV, *Analysis VIs*, describes the Analysis VIs. This part contains the following chapters:

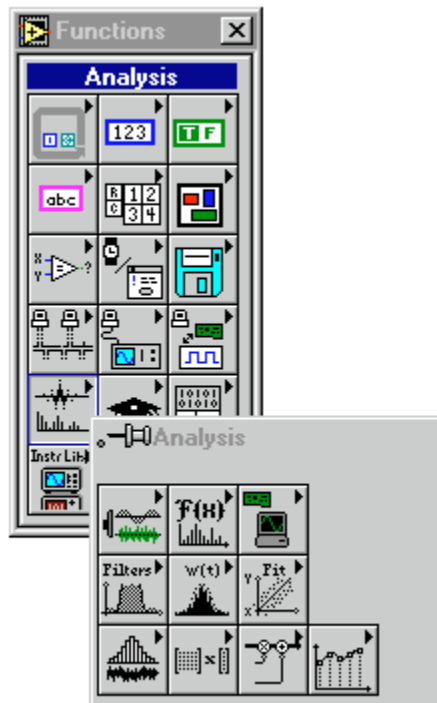
- Chapter 37, *Introduction to Analysis in LabVIEW*, introduces the LabVIEW Analysis VIs. It also provides a description of how the VIs are organized, instructions for accessing the VIs and obtaining online help, and a description of Analysis VI error reporting.
- Chapter 38, *Signal Generation VIs*, describes the VIs that generate one-dimensional arrays with specific waveform patterns.
- Chapter 39, *Digital Signal Processing VIs*, 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.
- Chapter 40, *Measurement VIs*, 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.
- Chapter 41, *Filter VIs*, describes the VIs that implement IIR, FIR, and nonlinear filters.
- Chapter 42, *Window VIs*, describes the VIs that implement smoothing windows.
- Chapter 43, *Curve Fitting VIs*, describes the VIs that perform curve fitting or regression analysis.
- Chapter 44, *Probability and Statistics VIs*, describes the VIs that perform probability, descriptive statistics, analysis of variance, and interpolation functions.
- Chapter 45, *Linear Algebra VIs*, describes the VIs that perform real and complex matrix related computation and analysis.

- Chapter 46, *Array Operation VIs*, describes the VIs that perform common, one- and two-dimensional numerical array operations.
- Chapter 47, *Additional Numerical Method VIs*, describes the VIs that use numerical methods to perform root-finding, numerical integration, and peak detection.

Introduction to Analysis in LabVIEW

This chapter introduces 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** palette from the block diagram window, choose **Functions»Analysis** and proceed through the hierarchical menus to select the VI you want. You can place the icon corresponding to that VI in the block diagram and then wire it.



Full Development System

The base Analysis VI library is a subset of the advanced analysis VI library available in the full development system. 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. After you upgrade, you have all the analysis tools available in the Full Development System.

Refer to Chapter 38 through Chapter 47, which introduce each analysis subpalette, 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. The **Analysis** palette includes the following subpalettes:

- **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 1D arrays (or vectors) and 2D arrays (or matrices).

Normal lower case letters represent scalars or constants. For example,

$$\begin{aligned} a, \\ \pi, \\ b = 1.234. \end{aligned}$$

Capital letters represent arrays. For example,

$$\begin{aligned} X, \\ A, \\ Y = a X + b. \end{aligned}$$

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.

$$X = \{x_0, x_1, x_2, \dots, x_{n-1}\}$$

The following scalar quantity represents the i^{th} element of the sequence X .

$$x_i, \quad 0 \leq i < n$$

The first element in the sequence is x_0 and the last element in the sequence is x_{n-1} , for a total of n elements.

The following sequence of numbers represents a 2D array containing n rows and m columns.

$$A = \begin{bmatrix} a_{00} & a_{01} & a_{02} & \cdots & a_{0m-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1m-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2m-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1m-1} \end{bmatrix}$$

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.

$$a_{ij}, 0 \leq i < n \text{ and } 0 \leq j < m$$

The first element in A is a_{00} and the last element is a_{n-1m-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

$$x_i = a, \text{ for } i = 0, 1, 2, \dots, n-1.$$

Multiplying the elements of an array by a scalar constant is represented by

$$Y = aX,$$

which corresponds to the sequence

$$Y = \{ax_0, ax_1, ax_2, \dots, ax_{n-1}\}$$

and is used instead of

$$y_i = ax_i, \text{ for } i = 0, 1, 2, \dots, n-1.$$

Similarly, multiplying a 2D array by a scalar constant is represented by

$$B = k A,$$

which corresponds to the sequence

$$B = \begin{bmatrix} ka_{00} & ka_{01} & ka_{02} & \dots & ka_{0m-1} \\ ka_{10} & ka_{11} & ka_{12} & \dots & ka_{1m-1} \\ ka_{20} & ka_{21} & ka_{22} & \dots & ka_{2m-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ ka_{n-10} & ka_{n-11} & ka_{n-12} & \dots & ka_{n-1m-1} \end{bmatrix}$$

and is used instead of

$$b_{ij} = ka_{ij}, \text{ for } i = 0, 1, 2, \dots, n-1 \text{ and } j = 0, 1, 2, \dots, m-1.$$

Empty arrays are possible in LabVIEW. An array with no elements is an empty array and is represented by

$$\text{Empty} = \text{NULL} = \emptyset = \{ \}.$$

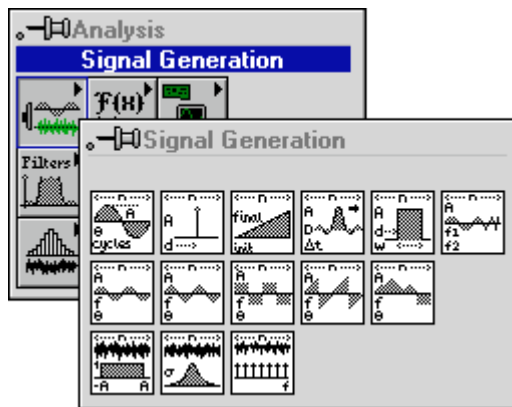
In general, operations on empty arrays result in empty, output arrays or undefined results.

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 Generation** palette, select **Function»Analysis»Signal Generation**. The following illustration shows the options that are available on the **Signal Generation** palette.



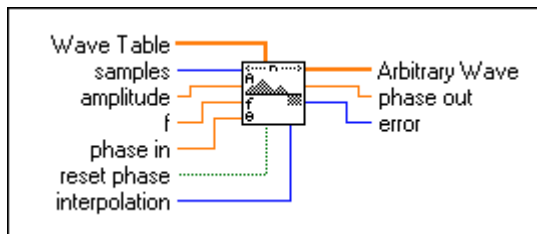
For examples of how to use the signal generation VIs, see the examples located in `examples\analysis\sigxmpl.llb`.

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:

$$y[i] = a * \text{arb}(\text{phase}[i]), \quad \text{for } i = 0, 1, 2, \dots, n-1,$$

where a is **amplitude**, n is the number of **samples**,

$$\text{arb}(\text{phase}[i]) = \text{WT}((\text{phase}[i] \bmod 360) * m / 360)$$

where m is the size of the **Wave Table** array.

If **interpolation** = 0 (no interpolation), then $\text{WT}(x) = \text{Wave Table}[\text{int}(x)]$.

If **interpolation** = 1 (linear interpolation), then $\text{WT}(x)$ is equal to the linearly interpolated value of **Wave Table** $[\text{int}(x)]$ and **Wave Table** $[(\text{int}(x)+1) \bmod m]$.

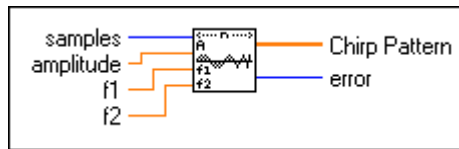
$\text{phase}[i] = \text{initial_phase} + f * 360.0 * i$, where f is the frequency in normalized units of cycles/sample, initial_phase is **phase in** if **reset phase** is true, or initial_phase is the **phase out** from the previous execution of this VI if **reset phase** is false.

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 $\text{phase}[n]$, 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.



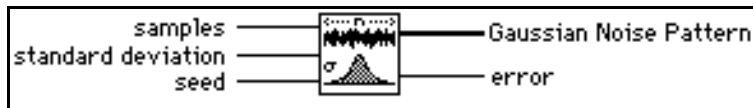
If the sequence Y represents **Chirp Pattern**, the VI generates the pattern according to the following formula:

$$y_i = A * \sin((a/2 i + b) i), \quad \text{for } i = 0, 1, 2, \dots, n-1,$$

where A is **amplitude**, $a = 2\pi(f_2 - f_1)/n$, $b = 2\pi f_1$, f_1 is the beginning frequency in normalized units of cycles/sample, f_2 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 $(\mu, \sigma) = (0, s)$, 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:

$$f(x) = \frac{1}{\sqrt{2\pi}s} e^{-\frac{1}{2}\left(\frac{x}{s}\right)^2},$$

where s is the absolute value of the specified **standard deviation** and that you can compute the expected values, $E\{\bullet\}$, 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:

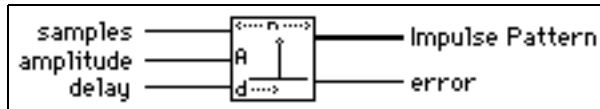
$$\mu = E\{x\} = 0,$$

$$\sigma = [E\{(x - \mu)^2\}]^{1/2} = s.$$

The pseudorandom sequence produces approximately 2^{90} samples before the pattern repeats itself.

Impulse Pattern

Generates an array containing an impulse pattern.



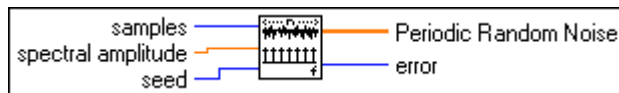
If **Impulse Pattern** is represented by the sequence X , the VI generates the pattern according to the following formula:

$$x_i = \begin{cases} a & \text{if } i = d \text{ for } i = 0, 1, 2, \dots, n-1 \\ 0 & \text{elsewhere} \end{cases}$$

where a is **amplitude**, d is **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.

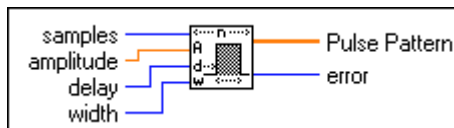
The output sequence is bounded by an amplitude of **spectral amplitude** * $\frac{\text{samples}}{2}$.

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.



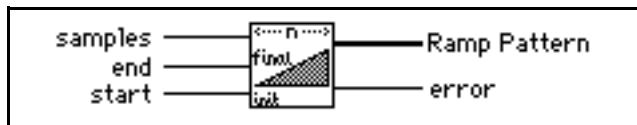
If the sequence X represents **Pulse Pattern**, the VI generates the pattern according to the following formula:

$$x_i = \begin{cases} a & \text{if } d \leq i < (d + w) \text{ for } i = 0, 1, 2, \dots, n-1. \\ 0.0 & \text{elsewhere} \end{cases}$$

where a is **amplitude**, d is **delay**, w is **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:

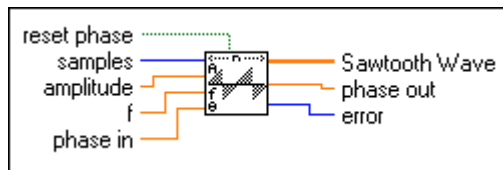
$$x_i = x_0 + i\Delta x \text{ for } i = 0, 1, 2, \dots, n-1,$$

where $\Delta x = \frac{x_{n-1} - x_0}{n - 1}$, x_{n-1} is **end**, x_0 is **start**, and n is the number of **samples**.

The VI 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:

$$y[i] = a * \text{sawtooth}(\text{phase}[i]), \text{ for } i = 0, 1, 2, \dots, n-1,$$

where a is the **amplitude**, n is the number of **samples**,

$$\text{sawtooth}(\text{phase}[i]) = \begin{cases} \frac{p}{180} & 0 \leq p < 180 \\ \frac{p}{180} - 2.0 & 180 \leq p < 360 \end{cases}$$

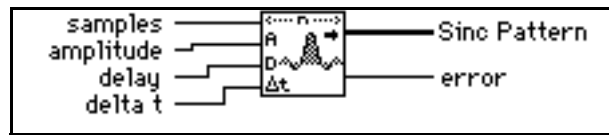
$p = \text{phase}[i] \text{ modulo } 360.0$, $\text{phase}[i] = \text{initial_phase} + \mathbf{f} * 360.0 * i$, \mathbf{f} is the frequency in normalized units of cycles/sample; initial_phase is **phase in** if **reset phase** is true; or initial_phase is the **phase out** from the previous execution 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 $\text{phase}[n]$, 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:

$$y_i = a \operatorname{sinc}(i\Delta t - d), \text{ for } i = 0, 1, 2, \dots, n-1,$$

where $\operatorname{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$, a is **amplitude**, Δt is the sampling interval **delta t**, d is

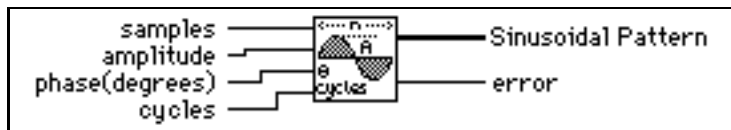
delay, and n is the number of **samples**.

The main lobe of the sinc function, $\operatorname{sinc}(x)$, is the part of the sinc curve bounded by the region $-1 \leq x \leq 1$.

When $|x| = 1$, the $\operatorname{sinc}(x) = 0.0$, and the peak value of the sinc function occurs when $x = 0$. Using l'Hôpital's Rule, you can show that $\operatorname{sinc}(0) = 1$ 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:

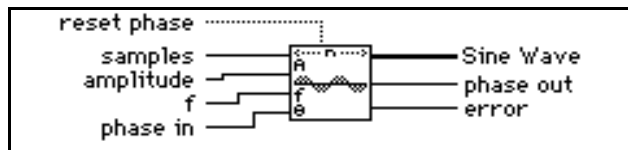
$$y_i = a \sin(x_i), \text{ for } i = 0, 1, 2, \dots, n-1,$$

where $x_i = \frac{2\pi i k}{n} + \frac{\pi\phi_0}{180}$, 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:

$$y_i = a * \sin(\text{phase}[i]), \text{ for } i = 0, 1, 2, \dots, n-1,$$

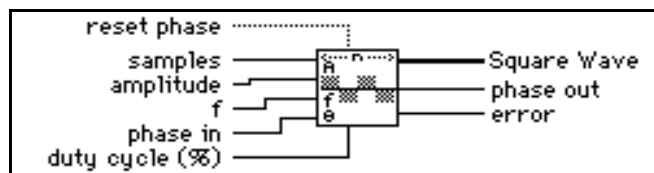
where a is **amplitude** and $\text{phase}[i] = \text{initial_phase} + f * 360.0 * i$; f is the frequency in normalized units of cycles/sample; initial_phase is **phase in** if **reset phase** is true; or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset 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 $\text{phase}[n]$, 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:

$$y_i = a * \text{square}(\text{phase}[i]), \text{ for } i = 0, 1, 2, \dots, n-1,$$

where a is **amplitude**; n is the number of **samples**;

$$\text{square}(\text{phase}[i]) = \begin{cases} 1.0 & 0 \leq p < \left(\frac{\text{duty}}{100}360\right) \\ -1.0 & \left(\frac{\text{duty}}{100}360\right) \leq p < 360, \end{cases}$$

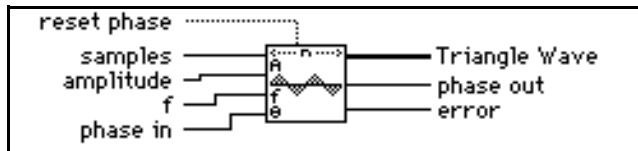
where $p = \text{phase}[i]$ modulo 360.0, $\text{duty} = \text{duty cycle}$, $\text{phase}[i] = \text{initial_phase} + \mathbf{f} * 360.0 * i$; \mathbf{f} is the frequency in normalized units of cycles/sample, initial_phase is **phase in** if **reset phase** is true; or initial_phase is the **phase out** 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 $\text{phase}[n]$, 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:

$$y_i = a * \text{tri}(\text{phase}[i]), \text{ for } i = 0, 1, 2, \dots, n-1$$

where a is **amplitude**; n is the number of **samples**;

$$\text{tri}(\text{phase}[i]) = \begin{cases} \frac{p}{90} & 0 \leq p < 90 \\ 2 - \frac{p}{90} & 90 \leq p < 270 \\ \frac{p}{90} + 4 & 270 \leq p < 360 \end{cases}$$

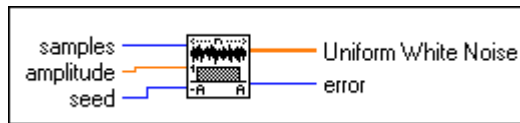
where $p = (\text{phase}[i] \text{ modulo } 360.0)$; $\text{phase}[i] = \text{initial_phase} + \mathbf{f} * 360.0 * i$; \mathbf{f} is the frequency in normalized units of cycles/sample; initial_phase is **phase in** if **reset phase** is true; or initial_phase is the **phase out** from the previous execution 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 $\text{phase}[n]$, 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

$$f(x) = \begin{cases} \frac{1}{2a} & \text{if } -a \leq x \leq a \\ 0 & \text{elsewhere} \end{cases}$$

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

$$\mu = E\{x\} = 0,$$

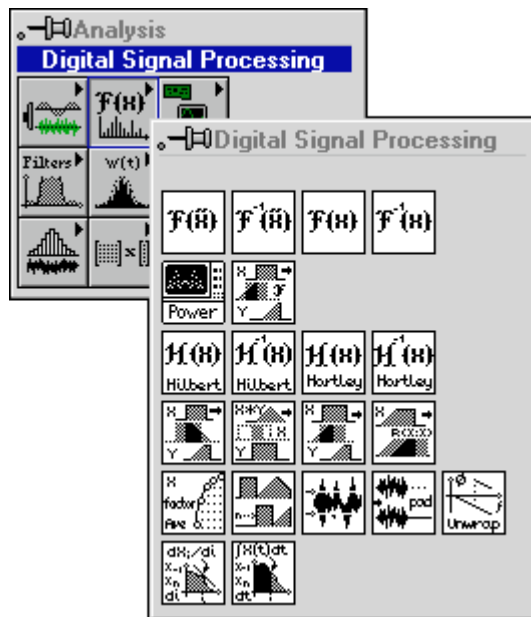
$$\sigma = [E\{(x - \mu)^2\}]^{1/2} = \frac{a}{\sqrt{3}} \approx 0.57735a.$$

The pseudorandom sequence produces approximately 2^{90} samples before the pattern repeats itself.

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 available on the **Digital Signal Processing** palette.



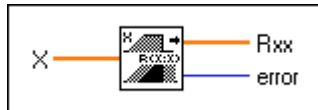
For examples of how to use the digital signal processing VIs, see the examples located in `examples\analysis\dspxmpl1.lib`.

Signal Processing VI Descriptions

The following Signal Processing VIs are available.

AutoCorrelation

Computes the autocorrelation of the input sequence \mathbf{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 \otimes 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 \mathbf{X} , and assume that the indexed elements of \mathbf{X} that lie outside its range are equal to zero,

$$x_j = 0, \quad j < 0 \quad \text{or} \quad j \geq n.$$

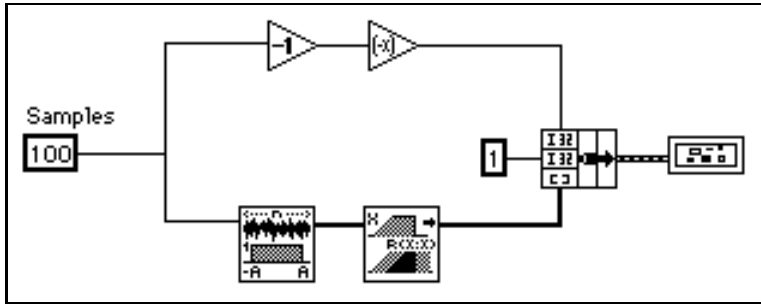
Then the VI obtains the elements of Y using

$$y_j = \sum_{k=0}^{n-1} x_k x_{j+k} \quad \text{for } j = -(n-1), -(n-2), \dots, -2, -1, 0, 1, 2, \dots, n-1.$$

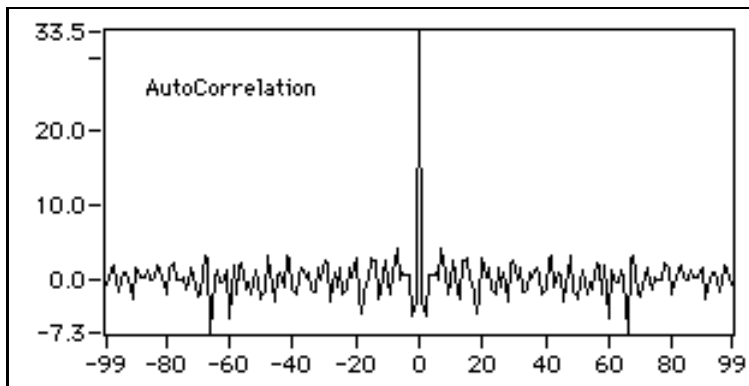
The elements of the output sequence \mathbf{Rxx} are related to the elements in the sequence Y by:

$$Rxx_i = y_{i-(n-1)} \quad \text{for } i = 0, 1, 2, \dots, 2n-2.$$

Notice that the number of elements in the output sequence \mathbf{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 \mathbf{Rxx} . Therefore, \mathbf{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.

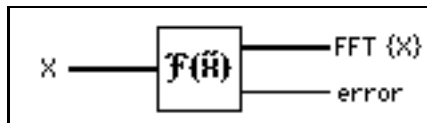


The following graph is the result of the preceding block diagram.



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 \mathbf{X} is a valid power of 2,

$$n = 2^m \text{ for } m = 1, 2, 3, \dots, 23,$$

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 \mathbf{X} is not a valid power of 2,

$$n \neq 2^m \text{ for } m = 1, 2, 3, \dots, 23,$$

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.*

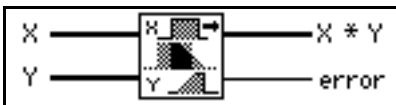
Let Y be the complex output sequence and n be the number of samples in it. It can be shown that

$$Y_{n-i} = Y_{-i}$$

which means you can interpret the $(n - i)^{th}$ element of Y as the $-i^{th}$ 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 \mathbf{X} and \mathbf{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 $\mathbf{X} * \mathbf{Y}$, let n be the number of elements in the input sequence \mathbf{X} , and let m be the number of elements in the input sequence \mathbf{Y} . Assuming that indexed elements of \mathbf{X} and \mathbf{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

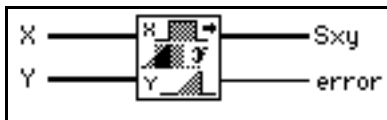
$$h_i = \sum_{k=0}^{n-1} x_k y_{i-k} \quad \text{for } i = 0, 1, 2, \dots, \text{size}-1,$$

size = $n + m - 1$,

where size denotes the total number of elements in the output sequence $\mathbf{X} * \mathbf{Y}$.

Cross Power

Computes the cross power spectrum of the input sequences \mathbf{X} and \mathbf{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

$$S_{xy} = \frac{1}{n} F^* \{X\} F \{Y\},$$

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,

$$n = m = 2^k \text{ for } k = 1, 2, 3, \dots, 23,$$

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,

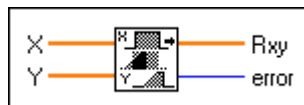
$$\max(n,m) = 2^k \text{ for } k = 1, 2, 3, \dots, 23,$$

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

$$\text{size} = \max(n,m).$$

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 \otimes 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 \mathbf{X} , let m be the number of elements in the sequence \mathbf{Y} , and assume that the indexed elements of \mathbf{X} and \mathbf{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

$$h_j = \sum_{k=0}^{n-1} x_k y_{j+k} \quad \text{for } j = -(n-1), -(n-2), \dots, -2, -1, 0, 1, 2, \dots, m-1.$$

The elements of the output sequence \mathbf{Rxy} are related to the elements in the sequence h by

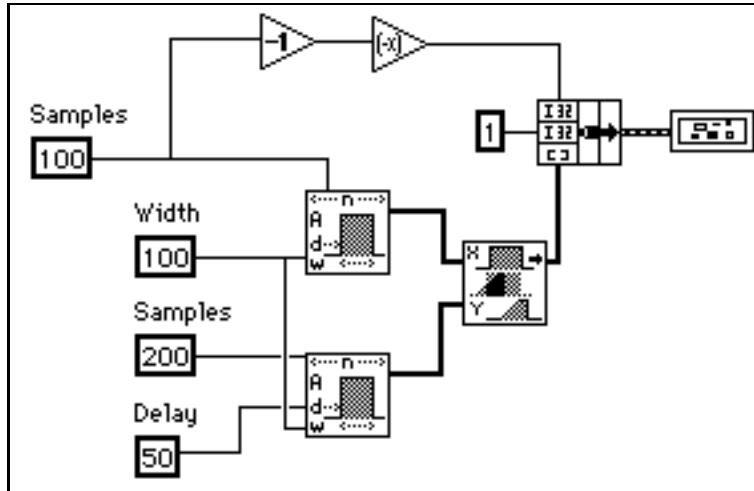
$$Rxy_i = h_{i-(n-1)} \quad \text{for } i = 0, 1, 2, \dots, \text{size}-1,$$

$$\text{size} = n + m - 1$$

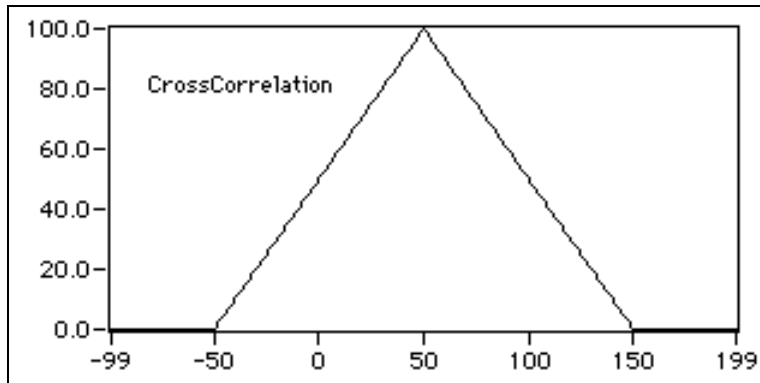
where size is the number of elements in the output sequence \mathbf{Rxy} .

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 \mathbf{Rxy} . Therefore, \mathbf{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.

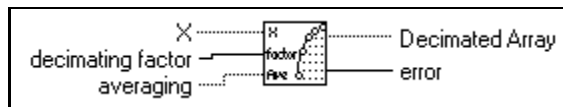


The following graph is the result of the preceding block diagram.



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

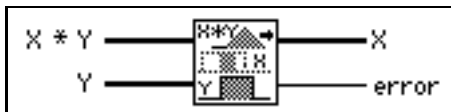
$$y_i = \begin{cases} x_{im} & \text{if ave is false} \\ \frac{1}{m} \sum_{k=0}^{m-1} x_{(im+k)} & \text{if ave is true} \end{cases} \quad \text{for } i = 0, 1, 2, \dots, \text{size}-1$$

$$\text{size} = \text{trunc}\left(\frac{n}{m}\right),$$

where n is the number of elements in \mathbf{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 $\mathbf{X} * \mathbf{Y}$ and \mathbf{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 $\mathbf{X} * \mathbf{Y}$.
2. Compute the Fourier transform of the input sequence \mathbf{Y} .
3. Divide the Fourier transform of $\mathbf{X} * \mathbf{Y}$ by the Fourier transform of \mathbf{Y} . Call the new sequence H .
4. Compute the inverse Fourier transform of H to obtain the deconvoluted sequence \mathbf{X} .

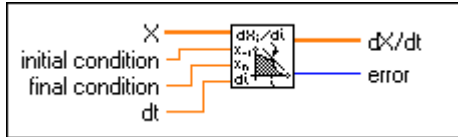


Note *The deconvolution operation is a numerically unstable operation, and it is not always possible to solve the system numerically. Computing the deconvolution via*

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 $\mathbf{dX/dt}$. The discrete implementation is given by

$$y_i = \frac{1}{2dt}(x_{i+1} - x_{i-1}) \text{ for } i = 0, 1, 2, \dots, n-1,$$

where n is the number of samples in $\mathbf{x(t)}$,

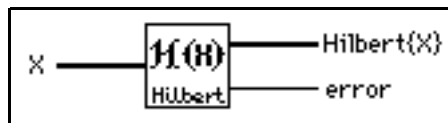
x_{-1} is specified by **initial condition** when $i = 0$, and

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

$$h(t) \Leftrightarrow H(f) = -j \operatorname{sgn}(f) X(f)$$

where $x(t) \Leftrightarrow X(f)$ is a Fourier transform pair and

$$\operatorname{sgn}(f) = \begin{cases} 1 & f > 0 \\ 0 & f = 0 \\ -1 & f < 0 \end{cases}$$

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 \mathbf{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: $Y_{\text{Nyq}} = 0$.
4. Multiply the positive harmonics by $-j$.
5. Multiply the negative harmonics by j . Call the new sequence H , which is of the form $H_k = -j \operatorname{sgn}(k) Y_k$.
6. Inverse Fourier transform H to obtain the Hilbert transform of \mathbf{X} .

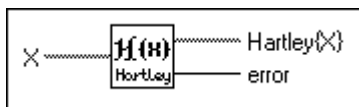
You use the Hilbert transform to extract instantaneous phase information, obtain the envelope of an oscillating signal, obtain single-sideband spectra, detect echoes, and reduce sampling rates.



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 \mathbf{X} .



The Hartley transform of a function $x(t)$ is defined as

$$X(f) = \int_{-\infty}^{\infty} x(t) \text{cas}(2\pi ft) dt$$

where $\text{cas}(x) = \cos(x) + \sin(x)$.

If Y represents the output sequence **Hartley{X}** obtained via the FHT, then Y is obtained through the discrete implementation of the Hartley integral:

$$Y_k = \sum_{i=0}^{n-1} X_i \text{cas}\left(\frac{2\pi ik}{n}\right), \text{ for } k = 0, 1, 2, \dots, n-1.$$

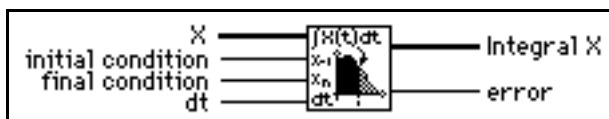
where n is the number of elements in \mathbf{X} .

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

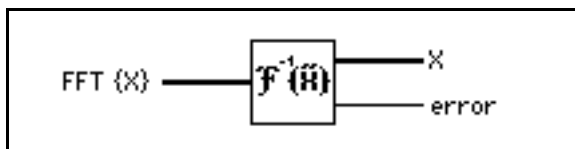
$$y_i = \frac{1}{6} \sum_{j=0}^i (x_{j-1} + 4x_j + x_{j+1}) dt \quad \text{for } i = 0, 1, 2, \dots, n-1,$$

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 $i = n - 1$.

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,

$$n = 2^m \quad \text{for } m = 1, 2, 3, \dots, 23,$$

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,

$$n \neq 2^m \quad \text{for } m = 1, 2, 3, \dots, 23,$$

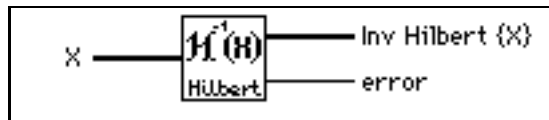
where n is the number of samples, the VI computes the inverse DFT by applying the chirp-z algorithm. The longest sequence with an inverse complex DFT that the VI can compute is $2^{22} - 1$ (4,194,303 or $4M - 1$).



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

$$h(t) = H^{-1}\{h(t)\} = \frac{1}{\pi} \int_{-\infty}^{\infty} \frac{h(\tau)}{t - \tau} d\tau.$$

Using the definition of the Hilbert transform

$$h(t) = H\{x(t)\} = -\frac{1}{\pi} \int_{-\infty}^{\infty} \frac{x(\tau)}{t - \tau} d\tau,$$

you obtain the inverse Hilbert transform by negating the forward Hilbert transform

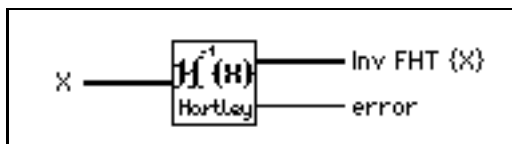
$$x(t) = H^{-1}\{h(t)\} = -H\{h(t)\}.$$

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 \mathbf{X} : $Y = H\{X\}$.
2. Negate Y to obtain the inverse Hilbert transform: $H^{-1}\{X\} = -Y$.

Inverse FHT

Computes the inverse fast Hartley transform (FHT) of the input sequence \mathbf{X} .



The inverse Hartley transform of a function $X(f)$ is defined as

$$x(t) = \int_{-\infty}^{\infty} X(f) \text{cas}(2\pi ft) df$$

where $\text{cas}(x) = \cos(x) + \sin(x)$.

If Y represents the output sequence **Inv FHT** $\{\mathbf{X}\}$, the VI calculates Y through the discrete implementation of the inverse Hartley integral:

$$Y_k = \frac{1}{n} \sum_{i=0}^{n-1} X_i \cos \frac{2\pi i k}{n}, \text{ for } k = 0, 1, 2, \dots, n-1.$$

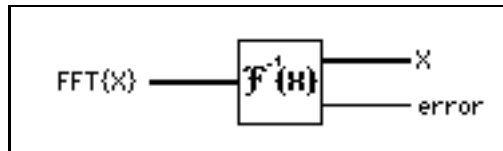
where n is the number of elements in \mathbf{X} .

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 earlier in this chapter 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** $\{\mathbf{X}\}$.



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:

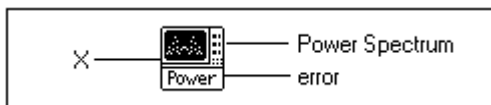
$$\text{size} = 2^m, m = 1, 2, \dots, 23.$$

If the size of the input sequence is not a power of 2, this VI calls an efficient Inverse DFT routine.

The output sequence $\mathbf{X} = \text{Inverse Real FFT} [\mathbf{FFT}\{\mathbf{X}\}]$ 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

$$S_{xx} = \frac{1}{n} |F\{\mathbf{X}\}|^2,$$

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,

$$n = 2^m \quad \text{for } m = 1, 2, 3, \dots, 23,$$

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,

$$n \neq 2^m \quad \text{for } m = 1, 2, 3, \dots, 23,$$

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 \mathbf{X} , and let n be the number of samples in it. It can be shown that

$$|Y_{n-i}|^2 = |Y_{-i}|^2.$$

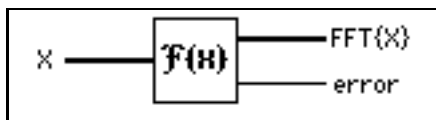
You can interpret the power in the $(n-i)^{\text{th}}$ element of Y as the power in the $-i^{\text{th}}$ element of the sequence, which represents the power in the *negative* i^{th} harmonic. You can find the total power for the i^{th} harmonic (DC and Nyquist component not included) using

$$\text{Power in } i^{\text{th}} \text{ harmonic} = 2|Y_i|^2 = |Y_i|^2 + |Y_{n-i}|^2, \quad 0 < i < \frac{n}{2}.$$

The total power in the DC and Nyquist components are $|Y_0|^2$ and $|Y_{n/2}|^2$, respectively.

Real FFT

Computes the Real Fast Fourier Transform (FFT) or the Real Discrete Fourier Transform (DFT) of the input sequence \mathbf{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:

$$\text{size} = 2^m, \quad m = 1, 2, \dots, 23.$$

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 $Y = \text{Real FFT}[\mathbf{X}]$ is complex and returns in one complex array:

$$Y = Y_{\text{Re}} + jY_{\text{Im}}$$

Unwrap Phase

Unwraps the **Phase** array by eliminating discontinuities whose absolute values exceed π .



$Y[i] = \text{Clip}\{X[i]\}$

Clips the elements of **Input Array** to within the bounds specified by **upper limit** and **lower limit**.



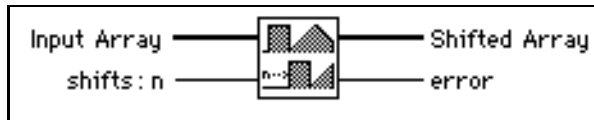
Let the sequence Y represent the output sequence **Clipped Array**; then the elements of Y are related to the elements of **Input Array** by

$$y_i = \begin{cases} a & x_i > a \\ x_i & b \leq x_i \leq a \\ b & x_i < b \end{cases} \text{ for } i = 0, 1, 2, \dots, n-1,$$

where n is the number of elements in **Input Array**, a is **upper limit**, and b is **lower limit**.

$Y[i] = X[i-n]$

Shifts the elements in **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

$$y_i = \begin{cases} x_{i - \text{shifts}} & \text{if } 0 \leq i - \text{shifts} < n \quad \text{for } i = 0, 1, 2, \dots, n-1, \\ 0 & \text{elsewhere} \end{cases}$$

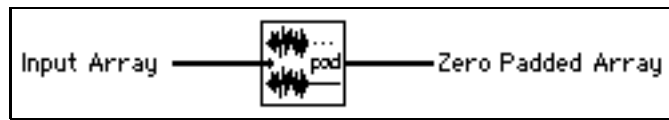
where n is the number of elements in **Input Array**.



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.

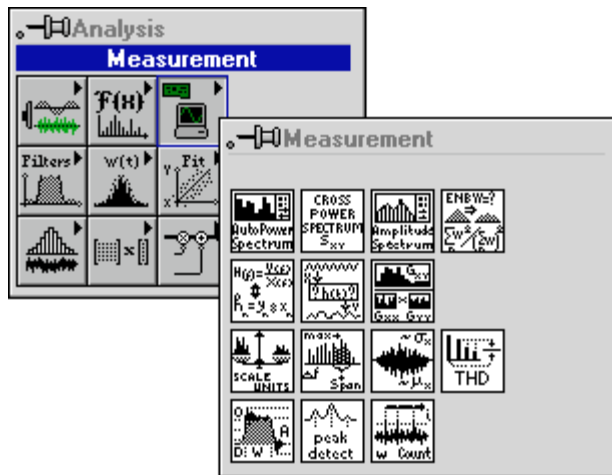


This VI is useful when the size of the acquired data buffers is not a power of 2, and you want to take advantage of fast processing algorithms in the analysis VIs. These algorithms include Fourier transforms, power spectrum, and FHTs, which are extremely efficient for buffer sizes that are a power of 2.

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 **Measurement** palette, select **Function»Analysis»Measurement**. The following illustration shows the options that are available on the **Measurement** palette.



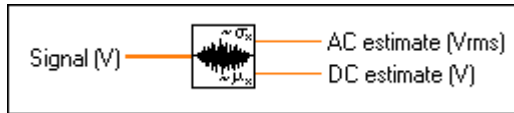
For examples of how to use the measurement VIs, see the examples using data acquisition located in `examples\analysis\measure\daqmeas.llb` and using simulated signals in `examples\analysis\measure\measxmpl.llb`.

Measurement VI Descriptions

The following Measurement VIs are available.

AC & DC Estimator

Computes an estimate 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.



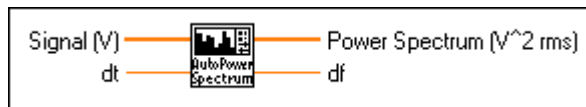
The VI computes the amplitude spectrum as

$$\frac{\text{FFT}(\text{Signal})}{N}$$

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.



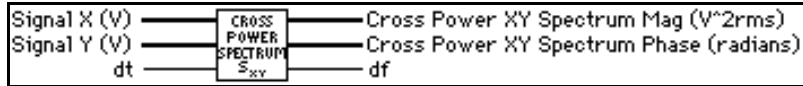
This VI computes the power spectrum as

$$\frac{\text{FFT}^*(\text{Signal}) \times \text{FFT}(\text{Signal})}{N^2}$$

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).



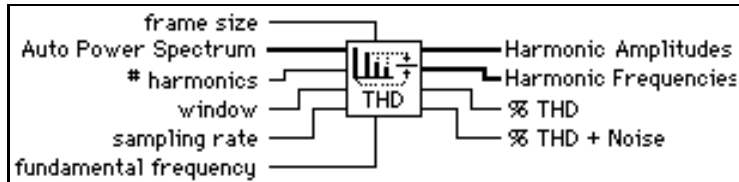
This VI computes the cross power spectrum as

$$\frac{\text{FFT}*(\text{Signal X}) \times \text{FFT}(\text{Signal Y})}{N^2}$$

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.

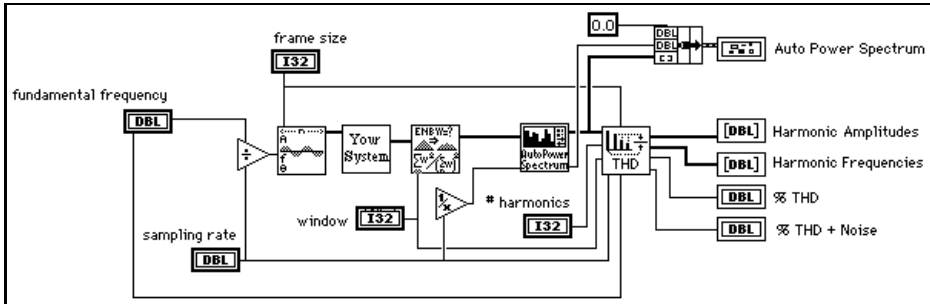
Harmonic Analyzer

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**).



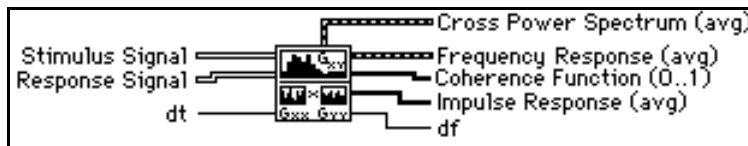
The **Impulse Response** is in the time domain, so you do not need to convert time units to frequency units. The **Impulse Response** is the inverse transform of the transfer function.

This VI computes **Impulse Response** as

$$\text{Inverse FFT} \left[\frac{\text{Cross Power}(\text{Stimulus}, \text{Response})}{\text{Power Spectrum}(\text{Stimulus})} \right]$$

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.

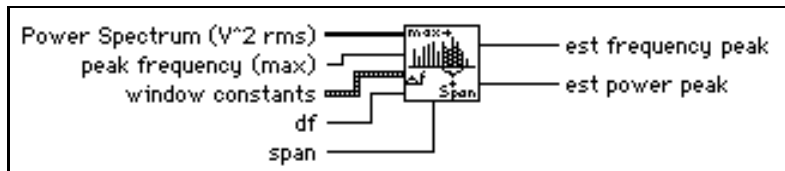
You can use this VI to measure the coherence between any two signals. The VI averages multiple stimulus and response signals to get valid coherence measurements. **Cross Power Spectrum** and **Impulse Response** are the rms averaged versions of the similarly named VIs. **Frequency Response** is the rms averaged version of the frequency response outputs of the Transfer Function VI.

Peak Detector

For information on this VI, see Chapter 47, [Additional Numerical Method VIs](#), in this manual.

Power & Frequency Estimate

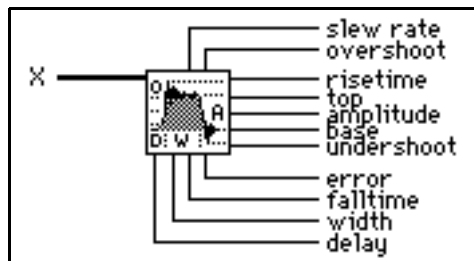
Computes the estimated power and frequency around a peak in the power spectrum of a time-domain signal.



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. Finds the maximum and minimum values in the input sequence **X**.
2. Generates the histogram of the pulse with 1% range resolution.
3. Determines the upper and lower modes to establish the **top** and **base** values.
4. Finds **overshoot**, **amplitude**, and **undershoot** from **top**, **base**, maximum, and minimum values.
5. Scans **X** and determines **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**.

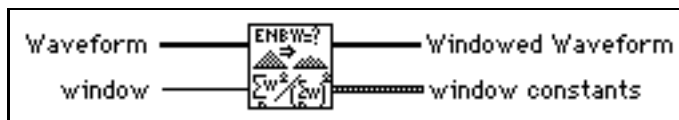
If **X** contains a train of pulses, the VI uses the train to determine **overshoot**, **top**, **amplitude**, **base**, and **undershoot**, but uses only the first pulse in the train to establish **slew rate**, **risetime**, **falltime**, **width**, and **delay**.



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

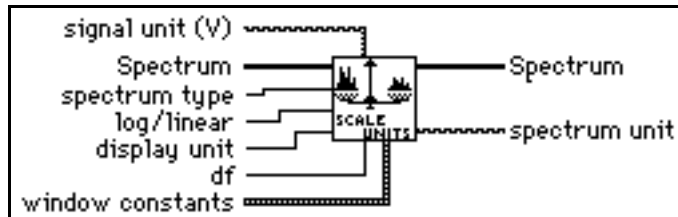
Applies the selected window to the time-domain signal.



The VI scales the result so that when the power or amplitude spectrum of **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.

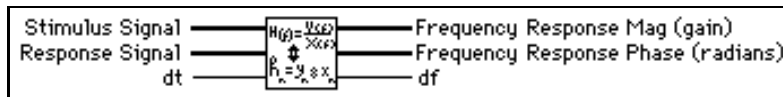


Threshold Peak Detector

For information on the this VI, see Chapter 47, [Additional Numerical Method VIs](#).

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.



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.

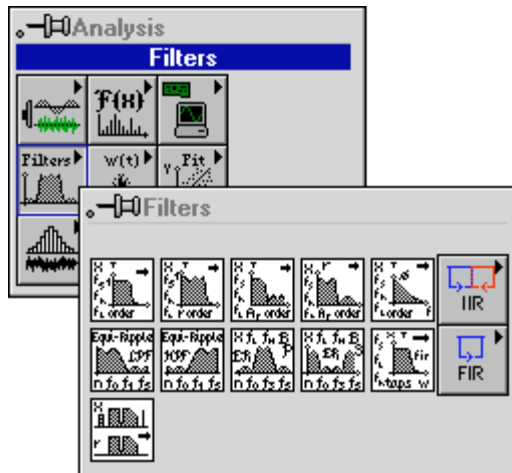
The VI computer frequency response is

$$\frac{\text{Cross Power}(\text{Stimulus}, \text{Response})}{\text{Power Spectrum}(\text{Stimulus})}$$

Filter VIs

This chapter describes the VIs that implement IIR, FIR, and nonlinear filters.

To access the **Filters** palette, select **Function»Analysis»Filters**. The following illustration shows the options that are available on the **Filters** palette.



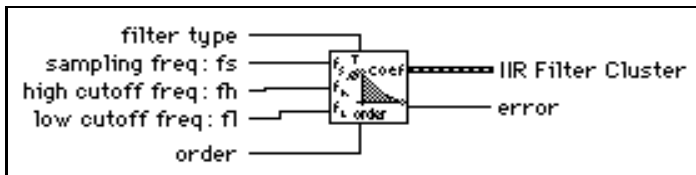
For examples of how to use the Filter VIs, see the examples located in `examples\analysis\fltrxmpl.llb`.

Filter VI Descriptions

The following Filter VIs are available.

Bessel Coefficients

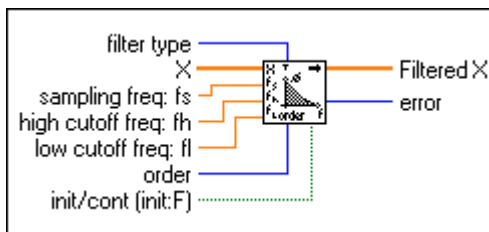
Generates the set of filter coefficients to implement an IIR filter as specified by the Bessel filter model. You can then pass these coefficients to the IIR Cascade Filter VI.



The Bessel Coefficients VI is a subVI of the Bessel Filter VI.

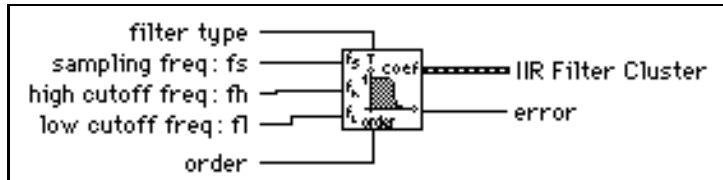
Bessel Filter

Generates a digital, Bessel filter using **filter type**, **sampling freq: fs**, **high cutoff freq: fh**, **low cutoff freq: fl**, and **order** by calling the Bessel Coefficients VI. The VI then calls the IIR Cascade Filter to filter the **X** sequence using this model to obtain a Bessel **Filtered X** sequence.



Butterworth Coefficients

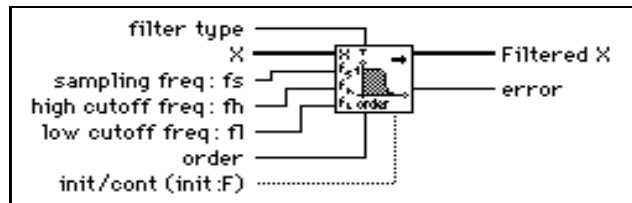
Generates the set of filter coefficients to implement an IIR filter as specified by the Butterworth filter model. You can pass these filter coefficients (**IIR Filter Cluster**) to the **IIR Cascade Filter VI** to filter a sequence of data.



This VI is a subVI of the Butterworth Filter VI.

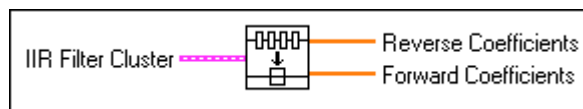
Butterworth Filter

Generates a digital Butterworth filter using **sampling freq: fs**, **low cutoff freq: fl**, **high cutoff freq: fh**, **order**, and **filter type** by calling the Butterworth Coefficients VI. The Butterworth Filter VI then calls the IIR Cascade 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:

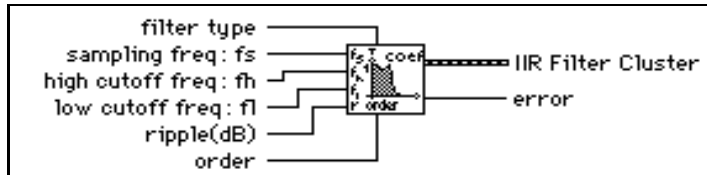
Reverse Coefficients: $\{a_{11}, a_{21}, a_{12}, a_{22}\} \rightarrow \{1.0, a_1, a_2, a_3, a_4\}$

Forward Coefficients: $\{b_{01}, b_{11}, b_{21}, b_{02}, b_{12}, b_{22}\} \rightarrow \{b_0, b_1, b_2, b_3, b_4\}$

See the IIR Cascade Filter VI for information about cascade form filtering, the IIR Filter VI for information on direct form filtering.

Chebyshev Coefficients

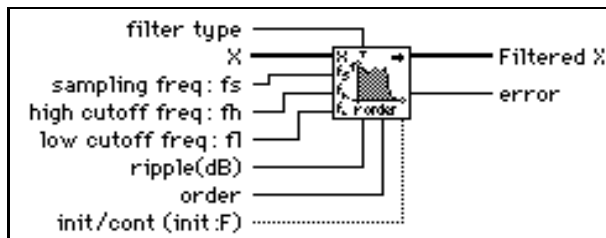
Generates the set of filter coefficients to implement an IIR filter as specified by the Chebyshev filter model. You can pass these coefficients to the IIR Cluster Filter VI to filter a sequence of data.



The Chebyshev Coefficients VI is a subVI of the Chebyshev Filter VI.

Chebyshev Filter

Generates a digital, Chebyshev filter using **sampling freq: fs**, **low cutoff freq: fl**, **high cutoff freq: fh**, **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 Cascade Filter VI.

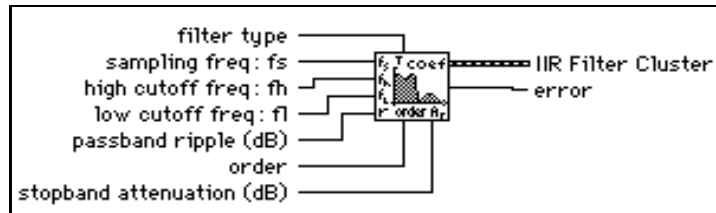


Convolution

For information about Convolution, see Chapter 39, [Digital Signal Processing VIs](#).

Elliptic Coefficients

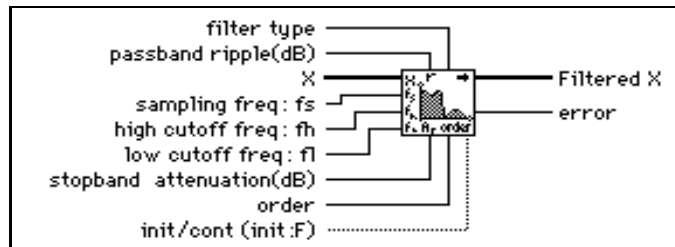
Generates the set of filter coefficients to implement a digital elliptic IIR filter. You can pass these coefficients to the IIR Cascade Filter VI.



The Elliptic Coefficients VI is a subVI of the Elliptic Filter VI.

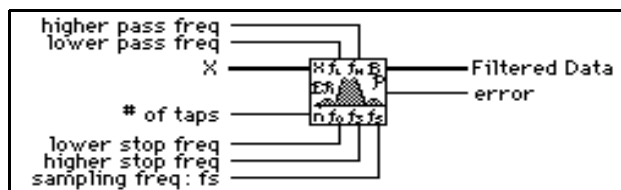
Elliptic Filter

Generates a digital, elliptic filter using **sampling freq: fs**, **low cutoff freq: fl**, **high cutoff freq: fh**, **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

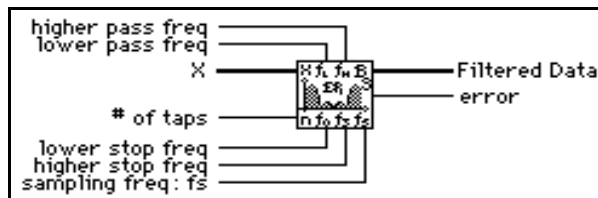
Generates a bandpass FIR filter with equi-ripple characteristics using the Parks-McClellan algorithm and **higher pass freq**, **lower pass freq**, **# of taps**, **lower stop freq**, **higher stop freq**, and **sampling freq: fs**. The VI then filters the input sequence **X** to obtain the bandpass, filtered, linear-phase sequence **Filtered Data**.



The first stopband of the filter region goes from zero (DC) to **lower stop freq**. The passband region goes from **lower pass freq** to **higher pass freq**, and the second stopband region goes from **higher stop freq** to the Nyquist frequency.

Equiripple BandStop

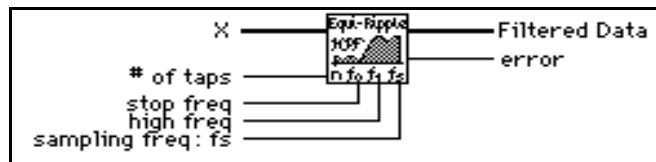
Generates a bandstop FIR digital filter with equi-ripple characteristics using the Parks-McClellan algorithm and **higher pass freq**, **lower pass freq**, **# of taps**, **lower stop freq**, **higher stop freq**, and **sampling frequency: fs**. The VI then filters the input sequence **X** to obtain the bandstop, filtered, linear-phase sequence **Filtered Data**.



The first passband region of the filter goes from zero (DC) to **lower pass freq**. The stopband region goes from **lower stop freq** to **higher stop freq**, and the second passband region goes from **higher pass freq** to the Nyquist frequency.

Equiripple HighPass

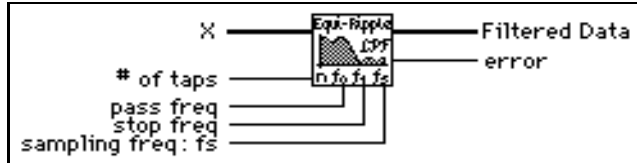
Generates a highpass FIR filter with equi-ripple characteristics using the Parks-McClellan algorithm and **# of taps**, **stop freq**, **high freq**, and **sampling freq**. The VI then filters the input sequence **X** to obtain the highpass, filtered, linear-phase sequence **Filtered Data**.



The stopband of the filter goes from zero (DC) to **stop freq**. The transition band goes from **stop freq** to **high freq**, and the passband goes from **high freq** to the Nyquist frequency.

Equiripple LowPass

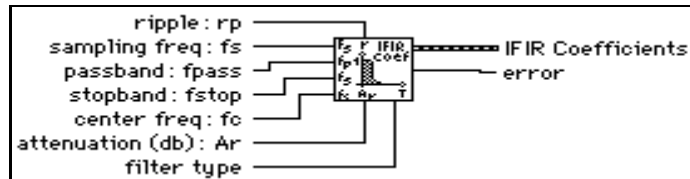
Generates a lowpass FIR filter with equiripple characteristics using the Parks-McClellan algorithm and the **# of taps**, **pass freq**, **stop freq**, and **sampling freq**. The VI then filters the input sequence **X** to obtain the lowpass filtered, linear-phase sequence **Filtered Data**.



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 passband ripple is shown as S_p . 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:

$$A_r = -20 \log \delta_A$$

$$\delta_A = 10^{\frac{-A_r}{20}}$$

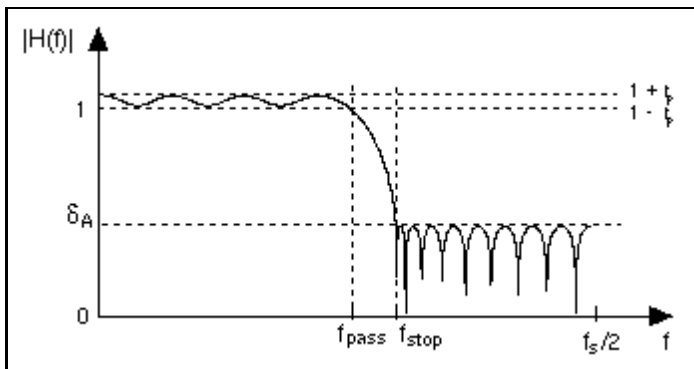


Figure 41-1. Lowpass Filter

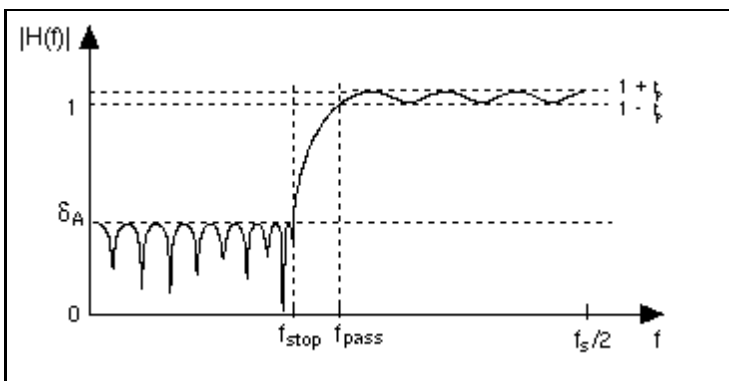


Figure 41-2. Highpass Filter

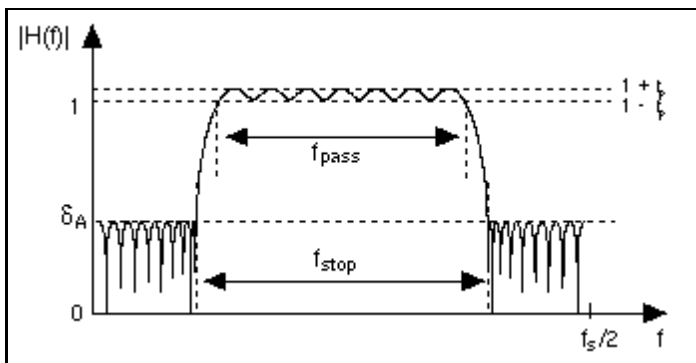


Figure 41-3. Bandpass Filter

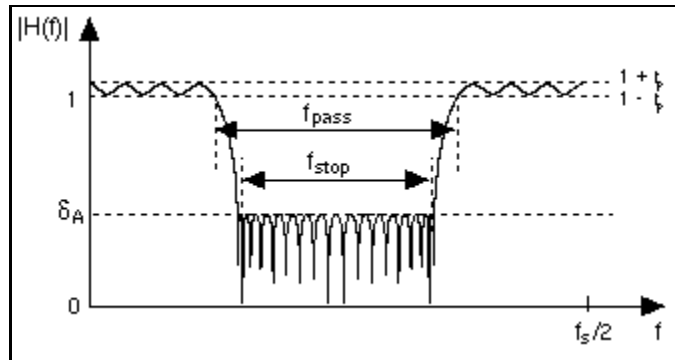
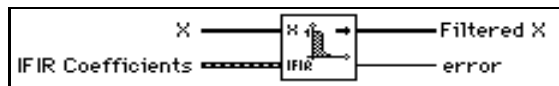


Figure 41-4. Bandstop Filter

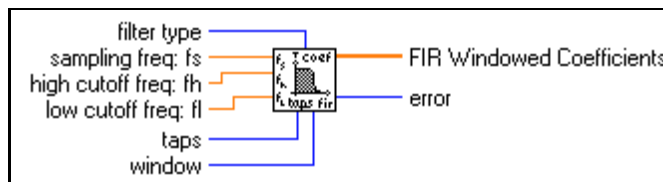
FIR Narrowband Filter

Filters the input sequence **X** using the IFIR filter specified by **IFIR Coefficients** as designed by the FIR Narrowband Filter Coefficients VI.



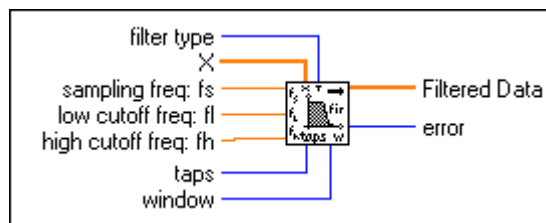
FIR Windowed Coefficients

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 **sampling freq: fs**, **low cutoff freq: fl**, **high cutoff freq: fh**, and number of **taps**.

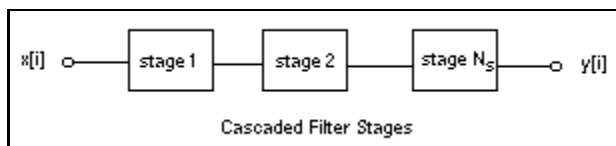


IIR Cascade Filter

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 $k = 1, 2, \dots, N_s$) 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 $2N_s$ and the total number of forward coefficients is $3N_s$. 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}, \dots, a_{4k}) and five forward coefficients (b_{0k}, \dots, b_{4k}).

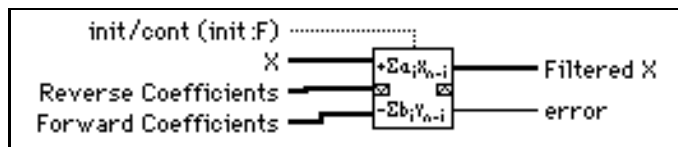
IIR Cascade Filter with Integrated Circuit

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**.



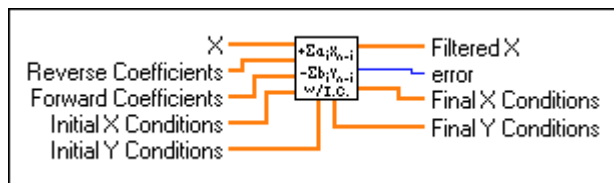
If y represents the output sequence **Filtered X**, the VI obtains the elements of y using

$$y_i = \frac{1}{a_0} \left(\sum_{j=0}^{n-1} b_j x_{i-j} - \sum_{k=1}^{m-1} a_k y_{i-k} \right),$$

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**.



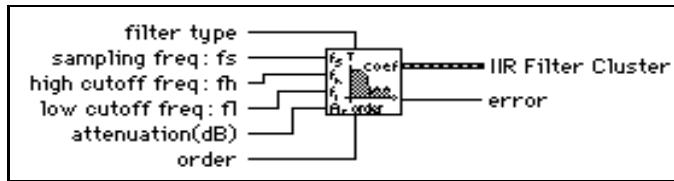
If y represents the output sequence **Filtered X**, the VI obtains the elements of y using

$$y_i = \frac{1}{a_0} \left(\sum_{j=0}^{n-1} b_j x_{i-j} - \sum_{k=1}^{m-1} a_k y_{i-k} \right),$$

where n is the number of **Forward Coefficients** (represented by b_j), and m is the number of **Reverse Coefficients** (represented by a_k).

Inv Chebyshev Coefficients

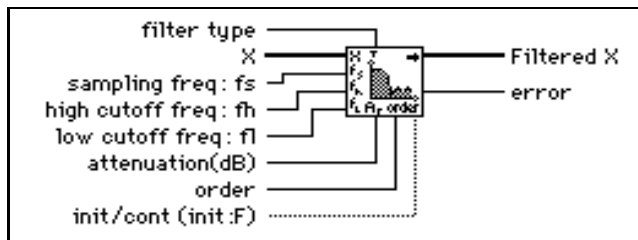
Generates the set of filter coefficients to implement an IIR filter as specified by the Chebyshev II Filter model. You can pass these coefficients to the IIR Cascade Filter VI to filter a sequence of data.



The Inv Chebyshev Coefficients VI is a subVI of the Inverse Chebyshev Filter VI.

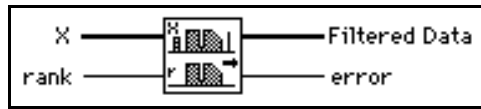
Inverse Chebyshev Filter

Generates a digital, Chebyshev II filter using the specified **sampling freq:fs**, **high cutoff freq: fh**, **low cutoff freq: fl**, **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**

$$J_i = \{x_{i-r}, x_{i-r+1}, \dots, x_{i-1}, x_i, x_{i+1}, \dots, x_{i+r-1}, x_{i+r}\},$$

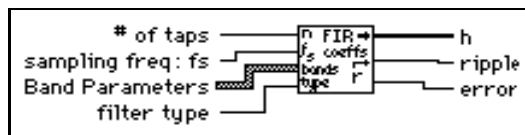
and if the indexed elements outside the range of **X** equal zero, the VI obtains the elements of y using

$$y_i = \text{Median}(J_i) \quad \text{for } i = 0, 1, 2, \dots, n - 1,$$

where n is the number of elements in the input sequence **X**, and r is the filter **rank**.

Parks-McClellan

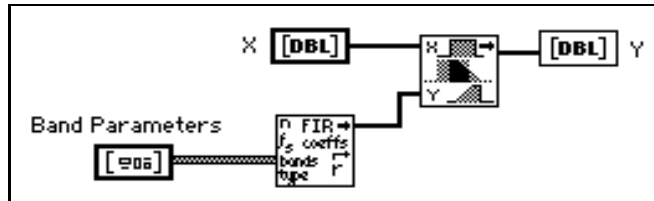
Generates a set of linear-phase FIR multiband digital filter coefficients using **# of taps**, **sampling freq: fs**, **Band Parameters**, and **filter type**.



Note

This VI finds the coefficients using iterative techniques based upon an error criterion. Although you specify valid filter parameters, the algorithm might fail to converge.

The Parks-McClellan 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.

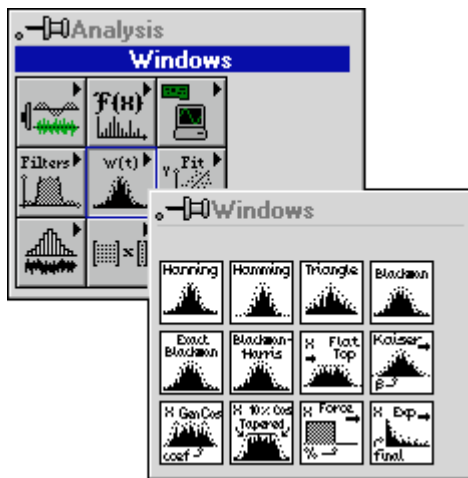


The equi-ripple filters use a similar technique to filter the data.

Window VIs

This chapter describes the VIs that implement smoothing windows.

To access the **Windows** palette, select **Function»Analysis»Windows**. The following illustration shows the options that are available on the **Windows** palette.



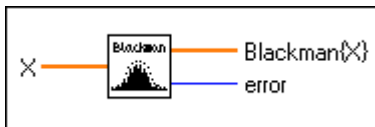
For examples of how to use the window VIs, see the examples located in `examples\analysis\windxmpl.llb`.

Window VI Descriptions

The following Window VIs are available.

Blackman Window

Applies a Blackman window to the input sequence \mathbf{X} .



If y represents the output sequence $\mathbf{Blackman}\{\mathbf{X}\}$, the VI obtains the elements of y from

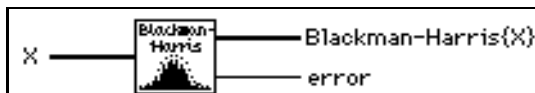
$$y_i = x_i [0.42 - 0.50 \cos(w) + 0.08 \cos(2w)] \quad \text{for } i = 0, 1, 2, \dots, n-1,$$

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in \mathbf{X} .

Blackman-Harris Window

Applies a three-term, Blackman-Harris window to the input sequence \mathbf{X} .



If y represents the output sequence $\mathbf{Blackman-Harris}\{\mathbf{X}\}$, the VI obtains the elements of y from

$$y_i = x_i [0.42323 - 0.49755 \cos(w) + 0.07922 \cos(2w)]$$

for $i = 0, 1, 2, \dots, n-1$

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in \mathbf{X} .

Cosine Tapered Window

Applies a cosine tapered window to the input sequence \mathbf{X} .



If y represents the output sequence **Cosine Tapered** $\{\mathbf{X}\}$, the VI obtains the elements of y from

$$y_i = \begin{cases} 0.5x_i(1 - \cos w) & \text{for } i = 0, 1, 2, \dots, m-1, \text{ and for } i = n-m, n-m+1, \dots, n-1 \\ x_i & \text{elsewhere} \end{cases}$$

$$\text{where } w = \frac{2\pi i}{n},$$

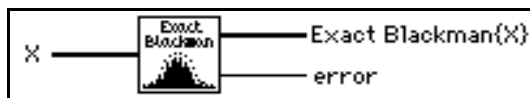
$$m = \text{round}\left(\frac{n}{10}\right), \text{ and}$$

where n is the number of elements in the input sequence \mathbf{X} .

Using this window is the equivalent of applying the Hanning window to the first and last 10% of the input sequence \mathbf{X} .

Exact Blackman Window

Applies an Exact Blackman window to the input sequence \mathbf{X} .



If y represents the output sequence **Exact Blackman** $\{\mathbf{X}\}$, the VI obtains the elements of y from

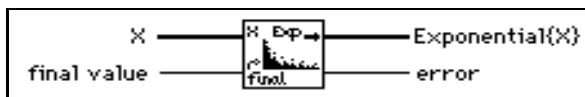
$$y_i = x_i [a_0 - a_1 \cos(w) + a_2 \cos(2w)] \text{ for } i = 0, 1, 2, \dots, n-1$$

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in \mathbf{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

$$y_i = x_i \exp(ai) \quad \text{for } i = 0, 1, 2, \dots, n-1,$$

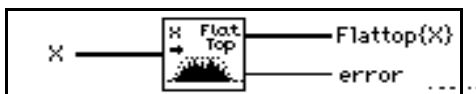
$$a = \frac{\ln(f)}{n-1},$$

where f is **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

$$y_i = x_i [0.2810639 - 0.5208972 \cos(w) + 0.1980399 \cos(2w)]$$

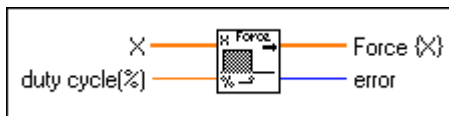
for $i = 0, 1, 2, \dots, n-1$

$$w = \frac{2\pi i}{n},$$

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

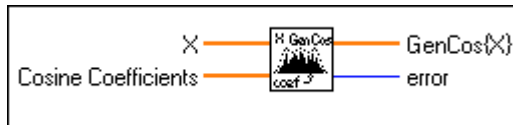
$$y_i = \begin{cases} x_i & \text{if } 0 \leq i \leq d \\ 0 & \text{elsewhere} \end{cases} \quad \text{for } i = 0, 1, 2, \dots, n-1$$

$d = (0.01)(n)(\text{duty cycle}(\%))$, 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

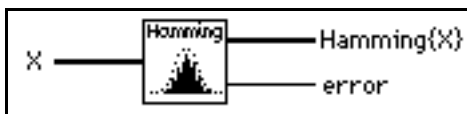
$$y_i = x_i \sum_{k=0}^{m-1} (-1)^k a_k \cos(kw) \quad \text{for } i = 0, 1, 2, \dots, n-1$$

$$w = \frac{2\pi i}{n},$$

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

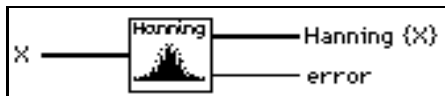
$$y_i = x_i [0.54 - 0.46 \cos(w)] \quad \text{for } i = 0, 1, 2, \dots, n-1,$$

$$w = \frac{2\pi i}{n},$$

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

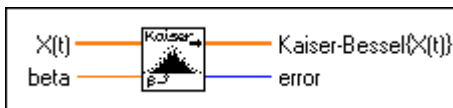
$$y_i = 0.5 x_i [1 - \cos(w)] \quad \text{for } i = 0, 1, 2, \dots, n-1,$$

$$w = \frac{2\pi i}{n},$$

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**{ $\mathbf{X}(t)$ }, the VI obtains the elements of y from

$$y_i = x_i \frac{I_0(\beta \sqrt{1.0 - a^2})}{I_0(\beta)} \text{ for } i = 0, 1, 2, \dots, n - 1$$

$$a = \frac{i - k}{k},$$

$$k = \frac{n - 1}{2},$$

where n is the number of elements in $\mathbf{X}(t)$, and $I_0(\bullet)$ is the zero-order modified Bessel function.

Triangle Window

Applies a triangular window to the input sequence \mathbf{X} .



Note *The Triangle smoothing window is also known as the Bartlett smoothing window.*

If y represents the output sequence **Triangle**{ \mathbf{X} }, the VI obtains the elements of y from

$$y_i = x_i \text{tri}(w) \text{ for } i = 0, 1, 2, \dots, n - 1,$$

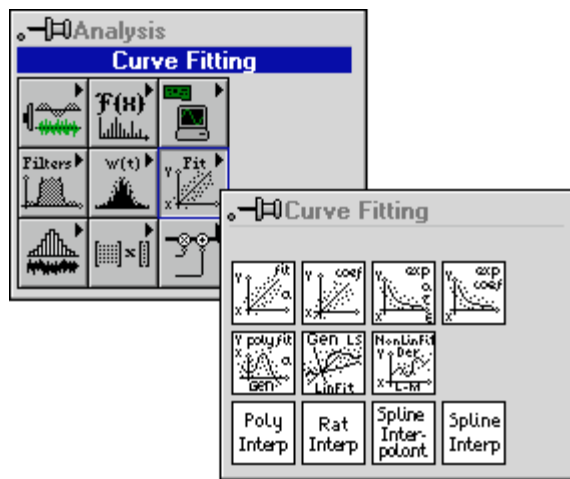
$$w = \frac{2i - n}{n},$$

where $\text{tri}(w) = 1 - |w|$, and n is the number of elements in \mathbf{X} .

Curve Fitting VIs

This chapter describes the VIs that perform curve fitting or regression analysis.

To access the **Curve Fitting** palette, choose **Functions»Analysis»Curve Fitting**, as shown in the following illustration.



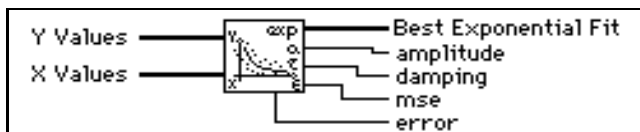
For examples of how to use the regression VIs, see the examples located in `examples\analysis\regressn.11b`.

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.

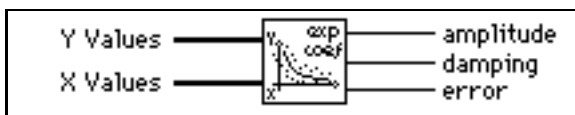
The VI obtains **mse** using the formula

$$\text{mse} = \frac{1}{n} \sum_{i=0}^{n-1} (f_i - y_i)^2,$$

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

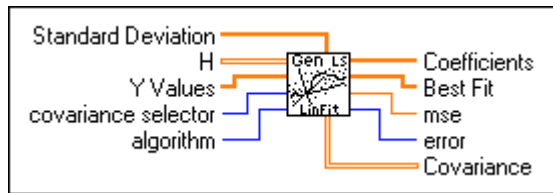
$$F = ae^{\tau X},$$

where F is the sequence representing the best fitted values, X represents the input sequence **X Values**, a is **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

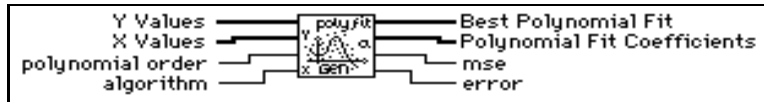
$\{x_{i0}, x_{i1}, \dots, x_{ik-1}, y_i\}$ where $i = 0, 1, \dots, n - 1$. n is the number of your observation data sets.



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.

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.



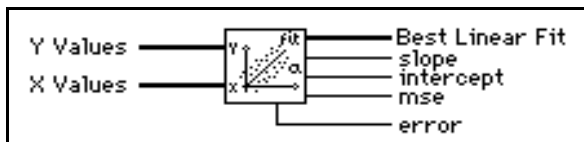
The general form of the polynomial fit is given by

$$f_i = \sum_{j=0}^m a_j x_i^j$$

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**.

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 **slope**, and b is **intercept**.

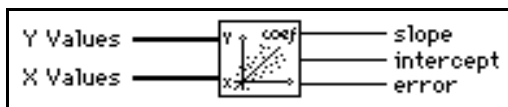
The VI obtains **mse** using the formula

$$\text{mse} = \frac{1}{n} \sum_{i=0}^{n-1} (f_i - y_i)^2,$$

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.

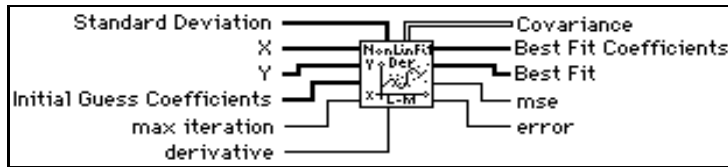
The general form of the linear fit is given by

$$F = mX + b,$$

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**.

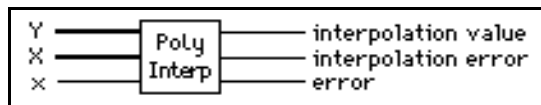
Nonlinear Lev-Mar Fit

Uses the Levenberg-Marquardt method to determine a nonlinear set of coefficients that minimize a chi-square quantity.



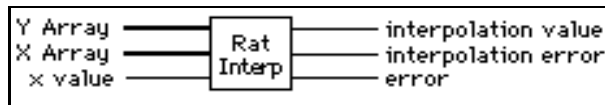
Polynomial Interpolation

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.



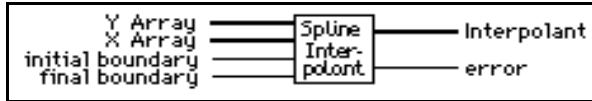
The rational function

$$\frac{P(x_i)}{Q(x_i)} = \frac{p_0 + p_1x_i + \dots + p_mx_i^m}{q_0 + q_1x_i + \dots + q_vx_i^v}$$

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 $i = 0, 1, \dots, n-1$. Input arrays **X Array** and **Y Array** are of length n and contain a tabulated function, $y_i = f(x_i)$, with $x_0 < x_1 < \dots < x_{n-1}$. **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 $i = 0, 1, \dots, n-1$.

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, $i = 0, 1, \dots, n-2$.

From the last condition, $p''_i(x_i) = p''_{i+1}(x_i)$, we derive the following equations:

$$\frac{x_i - x_{i-1}}{6} g''(x_{i-1}) + \frac{x_{i+1} - x_{i-1}}{3} g''(x_i) + \frac{x_{i+1} - x_i}{6} g''(x_{i+1})$$

$$= \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \quad i = 1, 2, \dots, n-2$$

These are $n-2$ linear equations with n unknowns $g''(x_i)$

$i = 0, 1, \dots, n-1$. This VI computes $g''(x_0)$, $g''(x_{n-1})$ from **initial boundary** and **final boundary** using the formula

$$g'(x) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + \frac{3A^2 - 1}{6}(x_{i+1} - x_i)g''(x_i) \\ + \frac{3B^2 - 1}{6}(x_{i+1} - x_i)g''(x_{i+1}).$$

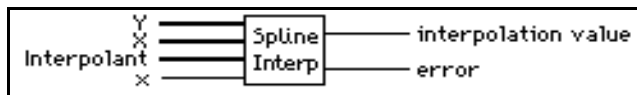
Here,

$$A = \frac{x_{i+1} - x}{x_{i+1} - x_i} \quad B = 1 - A = \frac{x - x_i}{x_{i+1} - x_i}$$

You can derive this formula from the preceding conditions. This VI then uses $g''(x_0)$, $g''(x_{n-1})$ to solve all the $g''(x_i)$, for $i = 1, \dots, n-2$. $g''(x_i)$ is the output **Interpolant**. You can use **Interpolant** as an input to the Spline Interpolation VI to interpolate y at any value of $x_0 \leq x \leq x_{n-1}$.

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 $i = 0, 1, \dots, n-1$, 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

$$y = Ay_i + By_{i+1} + 1 + Cy_i'' + Dy_{i+1}'' + 1,$$

and

$$A = \frac{x_{i+1} - x}{x_{i+1} - x_i},$$

$$B = 1 - A,$$

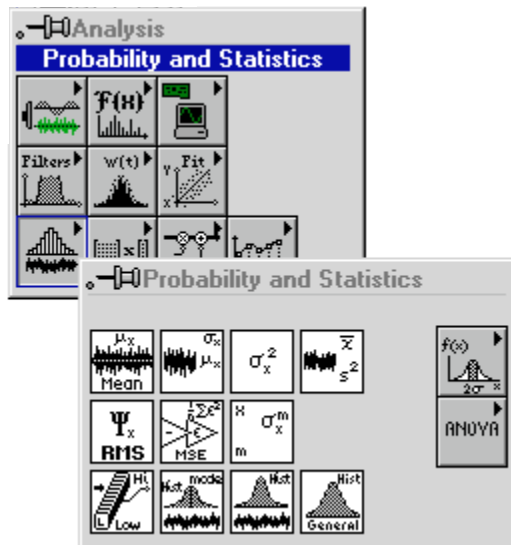
$$C = \frac{1}{6}(A^3 - A)(x_{i+1} - x_i)^2,$$

$$D = \frac{1}{6}(B^3 - B)(x_{i+1} - x_i)^2.$$

Probability and Statistics VIs

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

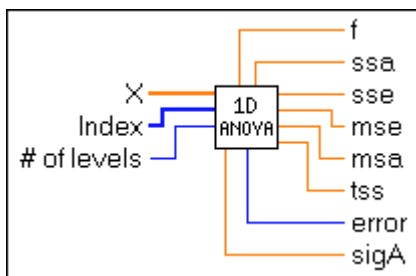
These VIs are not available in the Base Analysis package.

Probability and Statistics VI Descriptions

The following Probability and Statistics VIs are available.

1D ANOVA

Takes an array, \mathbf{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

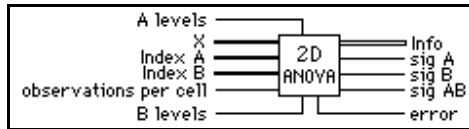
Notice that you have made at least one observation per level. To perform an analysis of variance, you must make at least one observation per level.

To perform the analysis of variance, you specify an array \mathbf{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)	76 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.

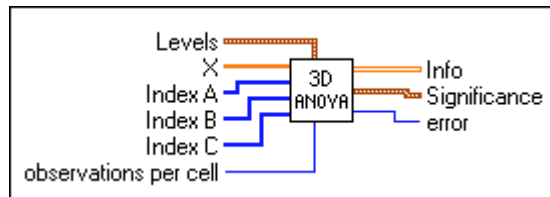
The array **Index B** specifies the level (or category) of factor B to which each observation applies. In this case, the array would have the values 0, 0, 2, 1, 1, and 2. Finally, there are two possible levels for factor A and three possible levels for factor B, so you pass in a value of 2 for the **A 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**:

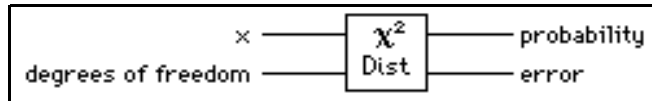
- Fixed-effects with interaction and $L > 1$ observations per cell.
- Any of the six mixed-effects models with interaction and $L > 1$ observations per cell.
- Random-effects with interaction and $L > 1$ observations per cell.

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**.

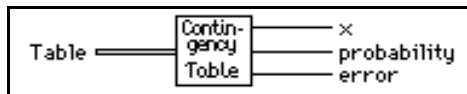


$$p = \text{Prob} \{X \leq x\},$$

where X is χ^2 distributed with n **degrees of freedom**, p is **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.



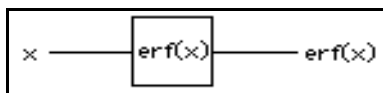
With the χ^2 test of homogeneity, the VI takes a random sample of some fixed size from each of the categories in one categorization scheme. For each of the samples, the VI categorizes the objects of experimentation according to the second scheme, and tallies them. The VI tests the hypothesis to determine 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.

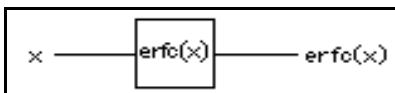
erf(x)

Evaluates the error function at the input value.



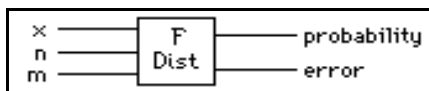
erfc(x)

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

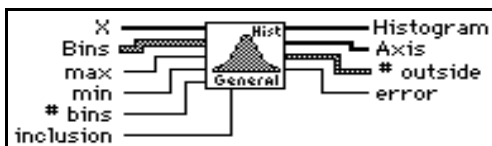


$$p = \text{Prob} \{F_{n,m} \leq x\},$$

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 **Histogram** as follows. First, the VI establishes all the intervals (also called bins) based on the information in the input array **Bins**. The intervals (bins) are:

$$\Delta_i = (\mathbf{Bins}[i].\text{lower} : \mathbf{Bins}[i].\text{upper}) \quad i = 0, 1, 2, \dots, k-1$$

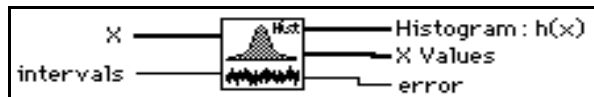
where

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) Δ_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

$$\mathbf{X} = \{0, 1, 3, 3, 4, 4, 4, 5, 5, 8\},$$

then the **Histogram: h(x)** of **X** for eight **intervals** is

$$h(\mathbf{X}) = \{h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7\} = \{1, 1, 0, 2, 3, 2, 0, 1\}.$$

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, Δx , 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

$$\chi_i = \min + 0.5\Delta x + i\Delta x \text{ for } i = 0, 1, 2, \dots, m-1.$$

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 $\chi_i + 0.5 \Delta x$,

$$\Delta_i = (\chi_i - 0.5 \Delta x : \chi_i + 0.5 \Delta x), \text{ for } i = 0, 1, 2, \dots, m-1,$$

and defines the function $y_i(x)$ to be

$$y_i(x) = \begin{cases} 1 & \text{if } x \in \Delta_i \\ 0 & \text{elsewhere} \end{cases}.$$

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 Δx .

The last interval, Δ_{m-1} , is defined as $[\chi_{m-1} - 0.5\Delta x : \chi_{m-1} + 0.5\Delta x]$. 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

$$h_i = \sum_{j=0}^{n-1} y_i(x_j) \text{ for } i = 0, 1, 2, \dots, m-1,$$

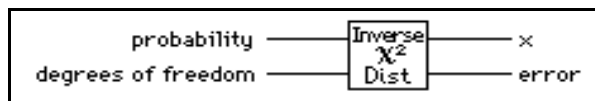
where h_i represents the elements of the output sequence **Histogram: h(x)**, and n is the number of elements in the input sequence **X**.

Inv Chi Square Distribution

Computes the value of **x** such that the condition

$$p = \text{Prob} \{X \leq \mathbf{x}\}$$

is satisfied, given the **probability** value, p , of a χ^2 -distributed random variable, X , with n **degrees of freedom**.

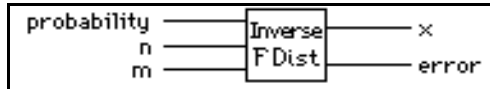


Inv F Distribution

Computes the value of x such that the condition

$$p = \{Prob_{n,m} \leq X\}$$

is satisfied, given the **probability** value p of an F-distributed random variable, F , with **n** and **m** degrees of freedom.

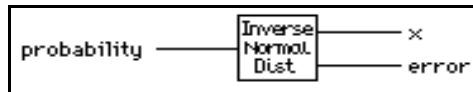


Inv Normal Distribution

Computes the value of x such that the condition

$$p = \text{Prob} \{X \leq x\}$$

is satisfied, given the **probability** value, p , of a Normally distributed random variable, X .

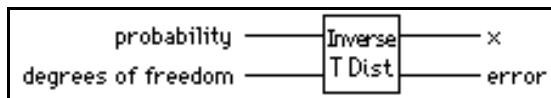


Inv T Distribution

Computes the value of x such that the condition

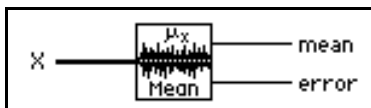
$$p = \text{Prob} \{T_n \leq x\}$$

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.



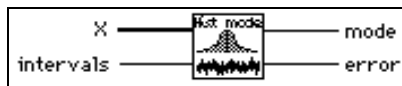
Let n be the number of elements in the input sequence **X**, and let S be the sorted sequence of **X**. The VI finds **median** using the following identity:

$$\text{median} = \begin{cases} s_i & \text{if } n \text{ is odd} \\ 0.5(s_{k-1} + s_k) & \text{if } n \text{ is even} \end{cases}$$

$$\text{where } i = \frac{n-1}{2}, \text{ and } k = \frac{n}{2}.$$

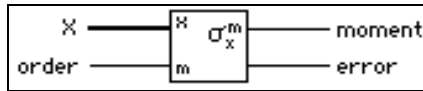
Mode

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**.



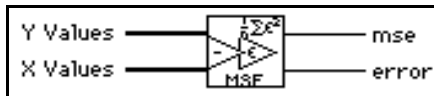
Let m be the desired **order**. The VI computes the m^{th} -order **moment** using the formula:

$$\sigma_x^m = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \mu)^m,$$

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, \mathbf{x} ,

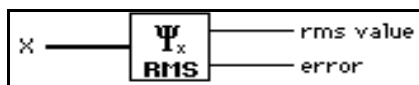
$$p = \text{Prob} \{X \leq \mathbf{x}\},$$

where X is standard Normally distributed, p is the **probability**, and \mathbf{x} is the value.



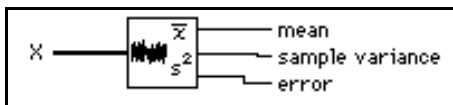
RMS

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**.

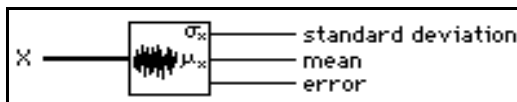


Note

*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:

$$\sigma_x = \sqrt{\frac{1}{n} \sum_{i=0}^{n-1} (x_i - \mu)^2},$$

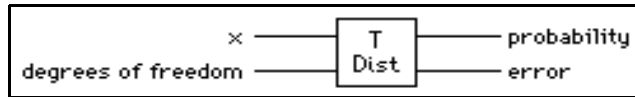
where $\mu = \frac{1}{n} \sum_{i=0}^{n-1} x_i$, and n is the number of elements in **X**.

T Distribution

Computes the one-sided **probability**, p , of the t-distributed random variable, T_n , with the specified **degrees of freedom**

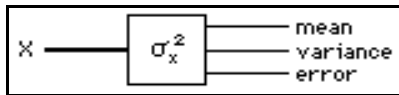
$$p = \text{Prob} \{T_n \leq x\},$$

where T is t-distributed, p is **probability**, n is **degrees of freedom**, and x is the value.



Variance

Computes the variance and the mean value of the input sequence \mathbf{X} .



This VI computes **variance** (σ_x^2) and **mean** (μ) using the following formula:

$$\sigma_x^2 = \frac{1}{n} \sum_{i=0}^{n-1} (x_i - \mu)^2,$$

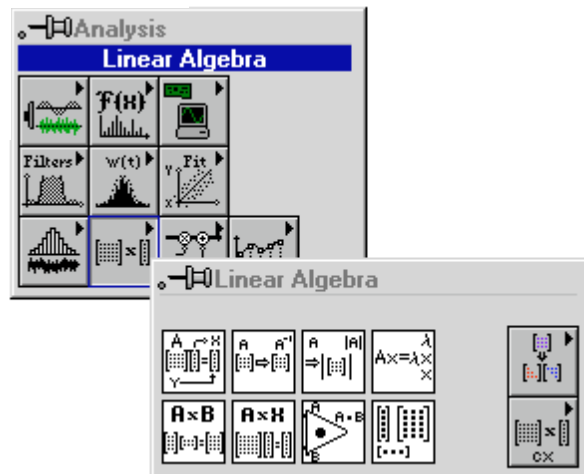
where $\mu = \frac{1}{n} \sum_{i=0}^{n-1} x_i$, and n is the number of elements in \mathbf{X} .

Linear Algebra VIs

This chapter describes the VIs that perform real and complex matrix related computation and analysis, including the following:

- Basic Matrix Manipulations
- Solving Linear Equations and Matrix Inverses
- Eigenvalues and Eigenvectors
- Matrix Analysis

To access the **Linear Algebra** palette, choose **Functions»Analysis»Linear Algebra**, as shown in the following illustration.



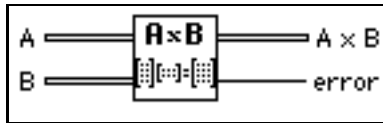
For examples of how to use the linear algebra VIs, see examples located in `examples\analysis\linaxmpl.llb`.

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 * B**. The VI obtains the elements of C using the formula

$$c_{ij} = \sum_{l=0}^{k-1} a_{il}b_{lj} \quad \text{for} \begin{cases} i = 0, 1, 2, \dots, n-1 \\ j = 0, 1, 2, \dots, m-1 \end{cases},$$

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.

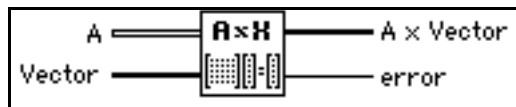


Note

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, $AB \neq BA$.

A x Vector

Performs the multiplication of an input matrix and an input vector.



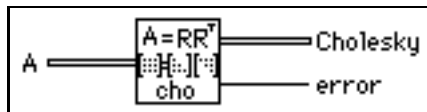
If A is an n -by- k matrix, and X is a vector with k elements, the multiplication of A and X , $Y = AX$, results in a vector Y with n elements. Let Y represent the output **A x Vector**. The VI obtains the elements of Y using the formula

$$y_i = \sum_{j=0}^{k-1} a_{ij}x_j, \text{ for } i = 0, 1, 2, \dots, n-1,$$

where n is the number of rows in \mathbf{A} , and k is the number of columns in \mathbf{A} and the number of elements in X .

Cholesky Factorization

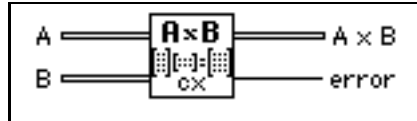
Performs Cholesky factorization for a real, positive definite matrix \mathbf{A} .



If the real, square matrix \mathbf{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 \mathbf{A} matrix, B represent the 2D input array \mathbf{B} matrix, and C represent the 2D output array $\mathbf{A} \times \mathbf{B}$. The VI obtains the elements of C using the formula

$$c_{ij} = \sum_{l=0}^{k-1} a_{il}b_{lj} \quad \text{for } \begin{cases} i = 0, 1, 2, \dots, n-1 \\ j = 0, 1, 2, \dots, m-1 \end{cases},$$

where n is the number of rows in \mathbf{A} matrix, k is the number of columns in \mathbf{A} matrix and the number of rows in \mathbf{B} matrix, and m is the number of columns in \mathbf{B} matrix.

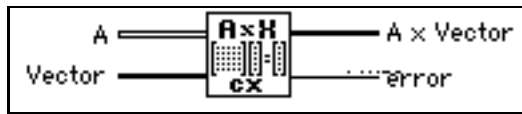


Note

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.



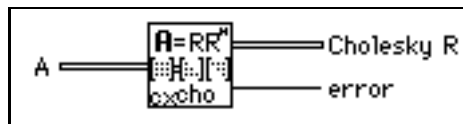
If \mathbf{A} is an n -by- k matrix, and X is a vector with k elements, the multiplication of \mathbf{A} and X , $Y = AX$, results in a vector Y with n elements. Let Y represent the output **A x Vector**, X represents the input vector. The VI obtains the elements of Y using the formula

$$y_i = \sum_{j=0}^{k-1} a_{ij}x_j, \text{ for } i = 0, 1, 2, \dots, n-1,$$

where n is the number of rows in \mathbf{A} , and k is the number of columns in \mathbf{A} and the number of elements in X .

Complex Cholesky Factorization

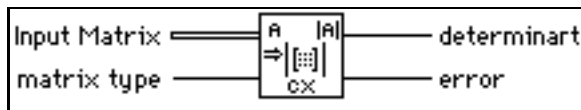
Performs Cholesky factorization of a complex, positive definite matrix \mathbf{A} .



If the complex square matrix \mathbf{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, respectively, of A such that

$$A = LU,$$

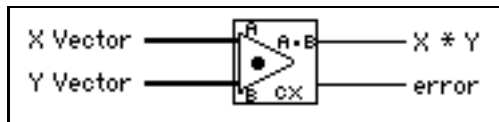
where the main diagonal elements of the lower triangular matrix L are arbitrarily set to one. The VI finds the **determinant** of A by the product of the main diagonal elements of the upper triangular matrix U :

$$|A| = \prod_{i=0}^{n-1} u_{ii},$$

where $|A|$ is the **determinant** of A , and n is the dimension of A .

Complex Dot Product

Computes the dot product of complex **X Vector** and **Y Vector**.



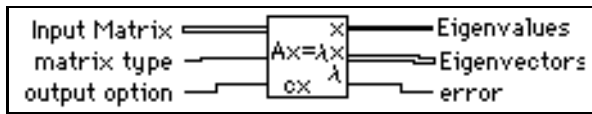
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains the dot product $X * Y$ using the formula:

$$X * Y = \sum_{i=0}^{n-1} x_i y_i,$$

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.



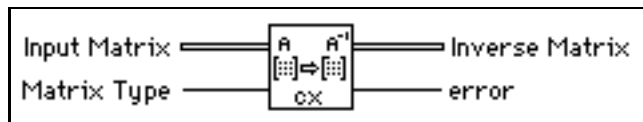
The eigenvalue problem is to determine the nontrivial solutions for the equation:

$$AX = \lambda X$$

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 **Eigenvectors** 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 Input Matrix and I be the identity matrix. You obtain **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

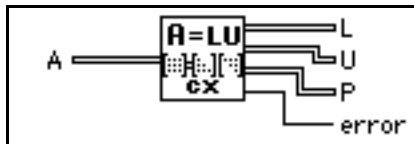
You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Complex Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.

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

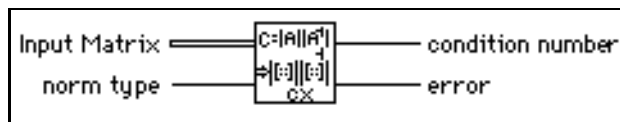
$$PA = LU$$

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

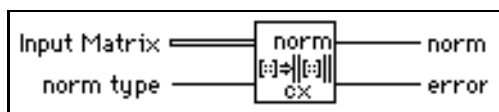
Computes the **condition number** of a complex matrix **Input Matrix**.



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

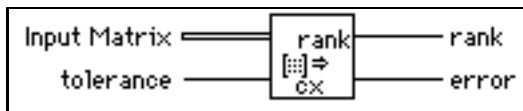
Computes the **norm** of a complex matrix **Input Matrix**.



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, ∞ . Different values of p mean different types of norms that are computed.

Complex Matrix Rank

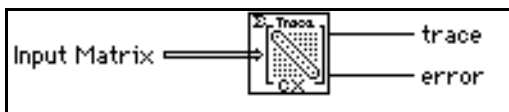
Computes the **rank** of a rectangular, complex matrix **Input Matrix**.



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**.



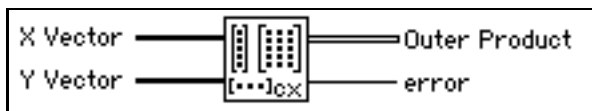
Let A be a square matrix that represents **Input Matrix** and $\text{tr}(A)$ be **trace**. The **trace** of A is the sum of the main diagonal elements of A

$$\text{tr}(A) = \sum_{i=0}^{n-1} a_{ii},$$

where n is the dimension of **Input Matrix**.

Complex Outer Product

Computes the outer product of a complex **X Vector** and **Y Vector**.



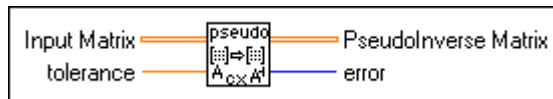
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains **Outer Product** using the formula:

$$a_{ij} = x_i y_j, \text{ for } \begin{cases} i = 0, 1, 2, \dots, n - 1 \\ j = 0, 1, 2, \dots, m - 1 \end{cases},$$

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**.

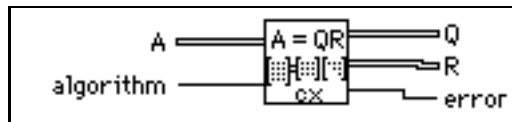


An SVD algorithm computes **PseudoInverse Matrix** A^+ , and treats any singular values less than the **tolerance** as zeros.

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 .

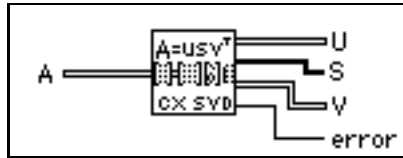


QR factorization is also called orthogonal-triangular factorization. It factors a complex matrix A into two matrices; one is an orthogonal matrix Q , and the other is an upper triangular matrix R , so that $A = QR$. This VI supports the Householder algorithm.

You can use QR factorization to solve linear systems that contain less or more equations than unknowns.

Complex SVD Factorization

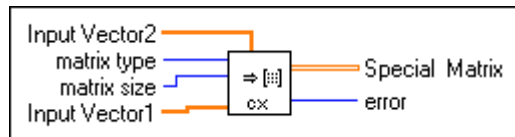
Performs the singular value decomposition (SVD) of a given m -by- n , complex matrix A with $m > n$.



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 .

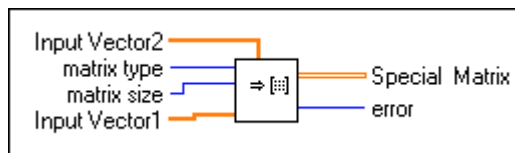
Create Special Complex Matrix

Generates a special, complex matrix based on **matrix type**. The available matrix types are Identity, Diagonal, Toeplitz, and Vandermonde.



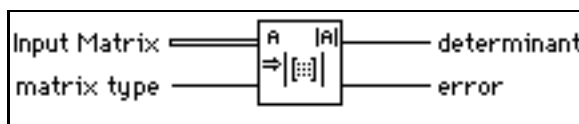
Create Special Matrix

Generates a real, special matrix based on **matrix type**. The available matrix types are Identity, Diagonal, Toeplitz, and Vandermonde.



Determinant

Computes the **determinant** of a real, square matrix **Input Matrix**.



Let A be a square matrix that represents **Input Matrix**, and let L and U represent the lower and upper triangular matrices, respectively, of A such that

$$A = LU,$$

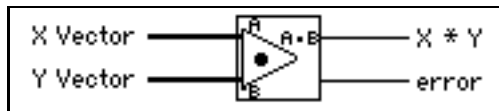
where the main diagonal elements of the lower triangular matrix L are arbitrarily set to one. The VI finds the **determinant** of A by the product of the main diagonal elements of the upper triangular matrix U

$$|A| = \prod_{i=0}^{n-1} u_{ii},$$

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**.



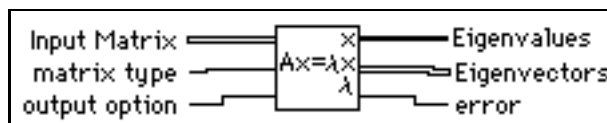
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains the dot product $X*Y$ using the formula:

$$X*Y = \sum_{i=0}^{n-1} x_i y_i,$$

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**.



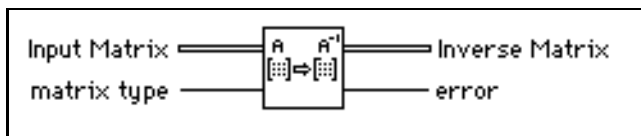
The eigenvalue problem is to determine the nontrivial solutions to the equation:

$$AX = \lambda X$$

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 \mathbf{A} be the **Input Matrix** and \mathbf{I} be the identity matrix. You obtain the **Inverse Matrix** value by solving the system $AB = I$ for B .

If \mathbf{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 \mathbf{A} :

$$B = A^{-1},$$

and \mathbf{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.



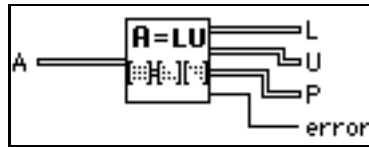
Note

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. The other is an upper triangular matrix **U**, so that

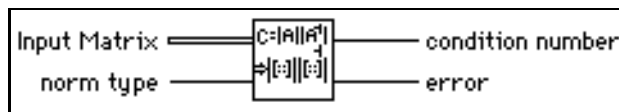
$$PA = LU,$$

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

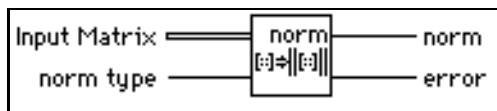
Computes the **condition number** of a real matrix **Input Matrix**.



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

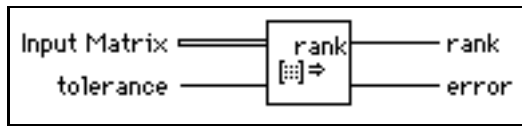
Computes the **norm** of a real matrix **Input Matrix**.



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

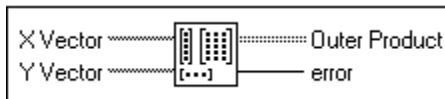
Computes the **rank** of a rectangular, real matrix **Input Matrix**.



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**.



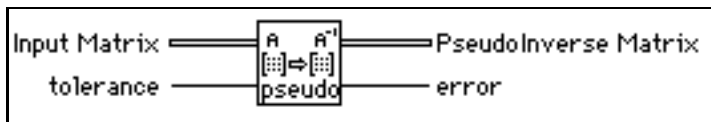
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains **Outer Product** using the formula

$$a_{ij} = x_i y_j, \text{ for } \begin{cases} i = 0, 1, 2, \dots, n - 1 \\ j = 0, 1, 2, \dots, m - 1 \end{cases},$$

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**.

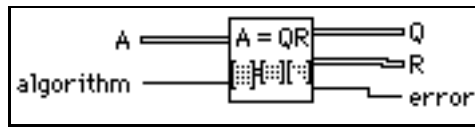


You compute **PseudoInverse Matrix** A^+ by using the SVD algorithm and any singular value less than the **tolerance**, which are set to zero.

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.

QR Factorization

Performs the QR factorization of a real matrix A .

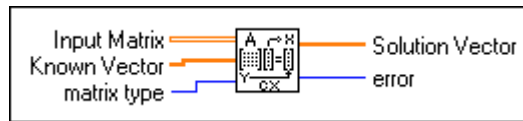


QR factorization is also called orthogonal-triangular factorization. It factors a real matrix A into two matrices. One is an orthogonal matrix Q , and the other is an upper triangular matrix R , so that $A = QR$. This VI provides three methods for the factorization: householder, givens, and fast givens.

You can use QR factorization to solve linear systems with more equations than unknowns.

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.

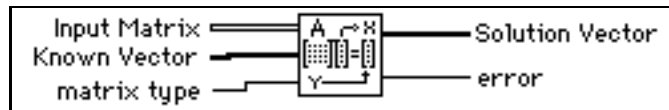


Note *You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.*

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.

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 $RX = Q^T Y$.

You can easily solve this triangular system to get x using recursive techniques.

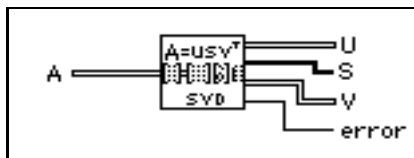


Note *You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.*

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 the system.

SVD Factorization

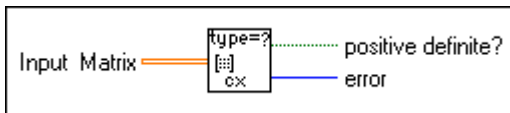
Performs the singular value decomposition (SVD) of a given m -by- n real matrix A , with $m > n$.



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 \mathbf{S} on the diagonal in decreasing order.

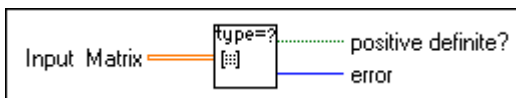
Test Complex Positive Definite

Tests whether **Input Matrix** is a Positive Definite matrix.



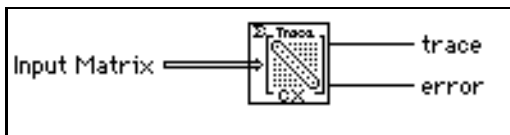
Test Positive Definite

Tests whether **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 $\text{tr}(A)$ be **trace**. The **trace** of A is the sum of the main diagonal elements of A

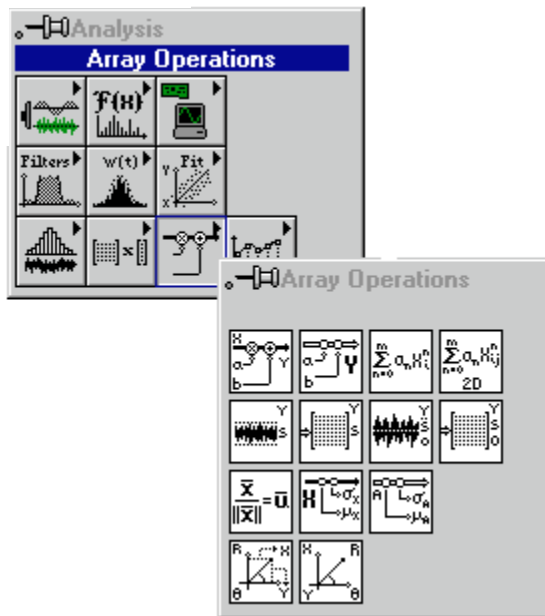
$$\text{tr}(A) = \sum_{i=0}^{n-1} a_{ii},$$

where n is the dimension of **Input Matrix**.

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**.

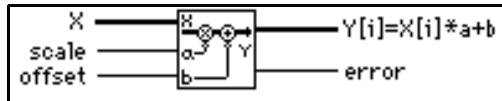


Array Operation VI Descriptions

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

$$Y = aX + b,$$

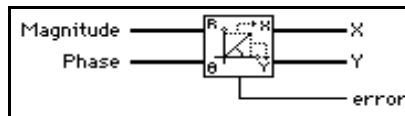
where a is the multiplicative **scale** constant, and b is the additive constant **offset**.

1D Polar To Rectangular

Converts two arrays of polar coordinates into two arrays of rectangular coordinates, according to the following formulas:

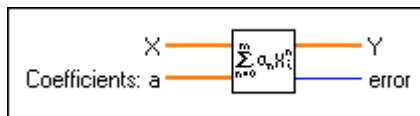
$$x = \text{Magnitude} \cos(\text{Phase})$$

$$y = \text{Magnitude} \sin(\text{Phase}).$$



1D Polynomial Evaluation

Performs a polynomial evaluation of **X** using **Coefficients: a**.



The output array \mathbf{Y} is given by

$$Y = \sum_{n=0}^m a_n X^n,$$

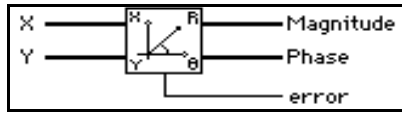
where m denotes the polynomial order.

1D Rectangular To Polar

Converts two arrays of rectangular coordinates into two arrays of polar coordinates, according to the following formulas:

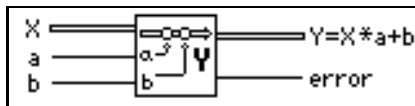
$$\text{magnitude} = \sqrt{x^2 + y^2}$$

$$\text{phase} = \tan^{-1} \left(\frac{y}{x} \right).$$



2D Linear Evaluation

Performs a linear evaluation of the two-dimensional input array \mathbf{X} .



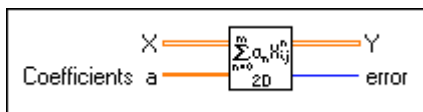
The two-dimensional output array $\mathbf{Y} = \mathbf{X} * \mathbf{a} + \mathbf{b}$ is given by

$$Y = Xa + b,$$

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**.



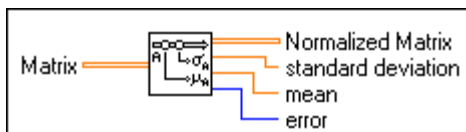
The 2D output array **Y** is given by

$$Y = \sum_{n=0}^m a_n X^n,$$

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

$$B = \frac{A - \mu}{\sigma},$$

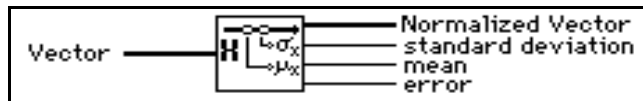
$$\mu = \frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_{ij}}{n \cdot m},$$

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} (a_{ij} - \mu)^2}{n \cdot m}},$$

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

$$Y = \frac{X - \mu}{\sigma},$$

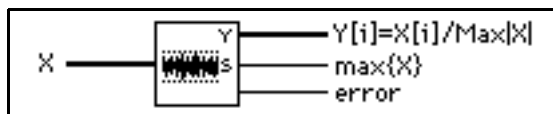
$$\mu = \frac{\sum_{i=0}^{n-1} x_i}{n},$$

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i - \mu)^2}{n}},$$

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 $Y[i] = X[i]/\text{Max}|X|$ is given by

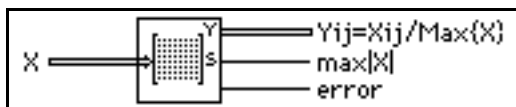
$$Y = \frac{X}{s},$$

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 $Y_{ij} = X_{ij}/\text{Max}\{X\}$ is given by

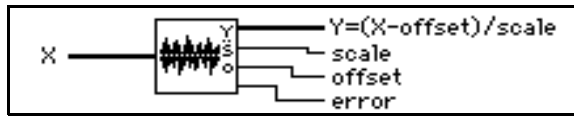
$$Y = \frac{X}{s},$$

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 matrix is a zero mean matrix.

Scale 1D

Determines **scale** and **offset** and then scales the input array **X** using these values.



The output array **Y** is given by

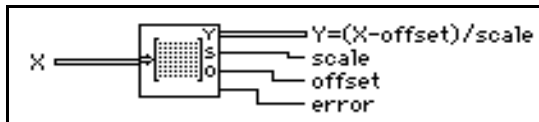
$$Y = \frac{X - \text{offset}}{\text{scale}},$$

scale = $0.5(\text{max} - \text{min})$, and **offset** = $\text{min} + \text{scale}$, 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** = $(\mathbf{X} - \text{offset})/\text{scale}$ is given by

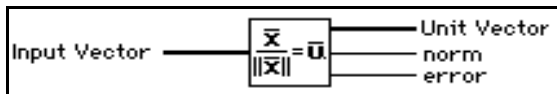
$$Y = \frac{X - \text{offset}}{\text{scale}},$$

scale = $0.5(\text{max} - \text{min})$, and **offset** = $\text{min} + 0.5 \text{ scale}$, 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

$$\|X\| = \sqrt{x_0^2 + x_1^2 + \dots + x_{n-1}^2},$$

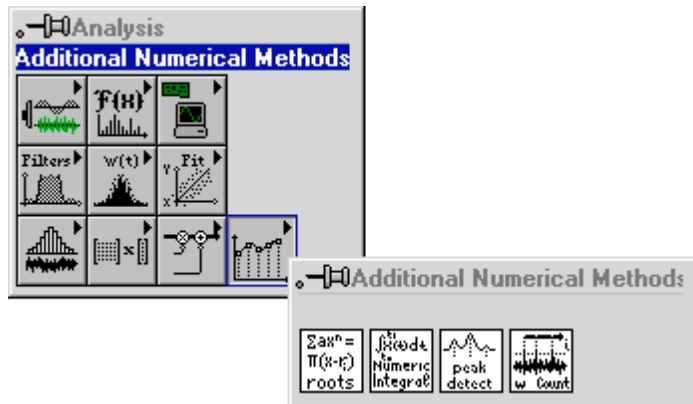
where $\|X\|$ is **norm**, and the VI calculates **Unit Vector**, U , using

$$U = \frac{X}{\|X\|}.$$

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**.

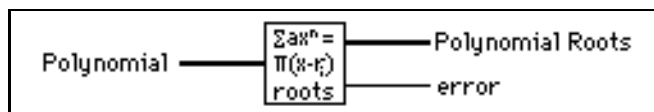


Additional Numerical Method VI Descriptions

The following Additional Numerical Method VIs are available.

Complex Polynomial Roots

Finds the complex roots of a complex polynomial.

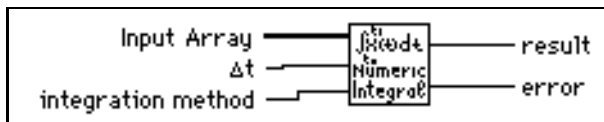


This VI uses a modified, complex Newton method to determine the n complex roots (some of which may be real, with a zero imaginary part), of the general complex polynomial:

$$a_0 + a_1x + a_2x^2 + \dots + a_{n-1}x^{n-1} + a_nx^n.$$

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.

$$\text{result} = \int_{t_0}^{t_1} f(t) dt \approx \sum_j \text{partial sums},$$

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': $(x[2i] + 4x[2i+1] + x[2i+2])*dt/3, k = 2$

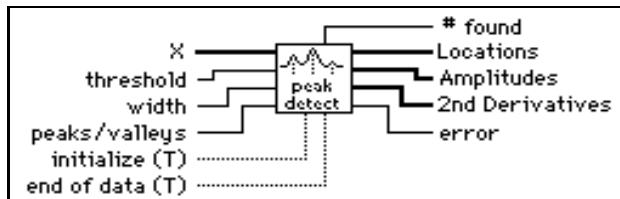
Simpsons' 3/8: $(3x[3i] + 9x[3i+1] + 9x[3i+2] + 3x[3i+3]) * dt/8, k = 3$

Bode: $(14x[4i] + 64x[4i+1] + 24x[4i+2] + 64x[4i+3] + 14x[4i+4])*dt/45, k = 4$
for $i = 0, k, 2k, 3k, 4k, \dots$, 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.

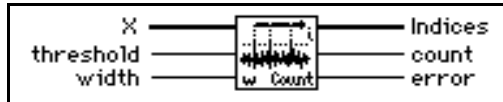
This VI is based on an algorithm that fits a quadratic polynomial to sequential groups of data points. The number of data points used in the fit is specified by **width**.

For each peak or valley, the quadratic fit is tested against the **threshold** level: peaks with heights lower than **threshold** or valleys with troughs higher than **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 **count** of the number of peaks encountered and a record of **Indices**, which locates the points that exceed **threshold** in a valid peak. A peak is valid where the elements of **X** exceed **threshold** and then return to a value less than or equal to **threshold**, and the number of elements that exceed **threshold** is at least equal to **width**.



Communication VIs and Functions

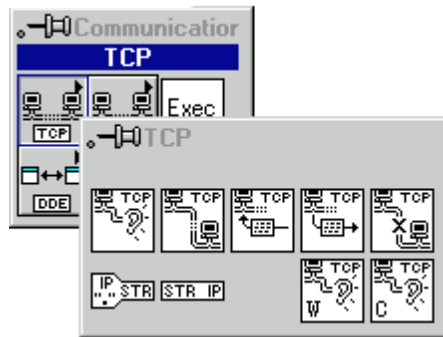
Part V, *Communication VIs and Functions*, describes how LabVIEW handles networking and interapplication communications and introduces the Communication VIs and functions. This part contains the following chapters:

- Chapter 48, *TCP VIs*, describes Internet Protocol (IP), Transmission Control Protocol (TCP), and internet addresses, and describes the LabVIEW TCP VIs. Refer to Chapter 21, *TCP and UDP VIs*, of the *LabVIEW User Manual* for an overview of TCP/IP and examples of TCP client/server applications.
- Chapter 49, *UDP VIs*, 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.
- Chapter 50, *DDE VIs*, 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.
- Chapter 51, *ActiveX Automation Functions*, describes the functions for support of ActiveX automation. These functions allow other ActiveX enabled applications, such as Microsoft Excel, to request properties and methods from LabVIEW and individual VIs.
- Chapter 52, *AppleEvent VIs*, describes the LabVIEW VIs for AppleEvents, one form of interapplication communication (IAC), through which Macintosh applications can communicate with each other.
- Chapter 53, *Program to Program Communication VIs*, 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.

TCP VIs

This chapter describes Internet Protocol (IP), Transmission Control Protocol (TCP), and internet addresses, and describes the LabVIEW TCP VIs. Refer to Chapter 21, *TCP and UDP*, of the *LabVIEW User Manual* 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**.



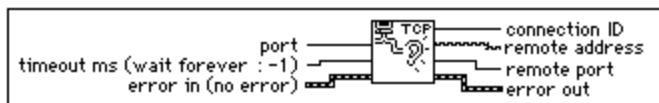
For examples of how to use the TCP VIs, see the examples in `examples\comm\tcpex.llb`.

TCP VI Description

The following TCP VI is available.

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 cannot 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/IP Functions

In addition to existing functions, some TCP/IP VIs are now functions. The following VIs are now functions in LabVIEW 5.0:

- IP To String
- String To IP
- TCP Open Connection
- TCP Create Listener
- TCP Wait on Listener
- TCP Write
- TCP Read
- TCP Close Connection

The TCP Listen VI is still a VI in LabVIEW 5.0 because its functionality is duplicated by the TCP Create Listener and the TCP Wait on Listener functions.

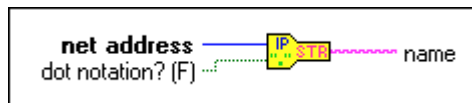
The TCP Read, TCP Write, and TCP Wait On Listener functions incorporate new functionality. TCP Write's **data in** parameter now accepts arrays of bytes. TCP Read has a new input, **mode**, which affects how it operates. The four modes are Standard, Buffered, CRLF, and Immediate. TCP Wait On Listener has a new input, **resolve remote address**, that tells whether to resolve the remote address or leave it in dot notation.

Standard has the same behavior it had in earlier versions of LabVIEW. Buffered is an all-or-nothing read. If you have not received the bytes requested at the end of a timeout, no bytes are returned. The unreturned bytes are saved for later read attempts. CRLF is read until a carriage return and linefeed is found in the input stream. You still must specify a maximum read size. If the CRLF is not found within the size expressed, nothing is returned. If the timeout limit is reached and a CRLF is not found, nothing is returned. Immediate specifies to return immediately from a read when any bytes are received.

The following TCP/IP functions are available.

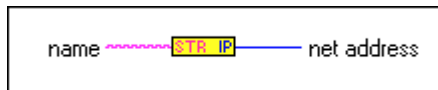
IP To String

Converts an IP network address to a string.



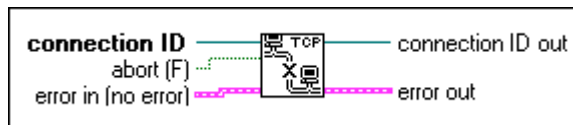
String To IP

Converts a string to an IP network address.



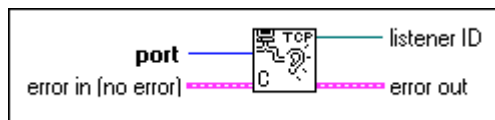
TCP Close Connection

Closes the connection associated with **connection ID**.



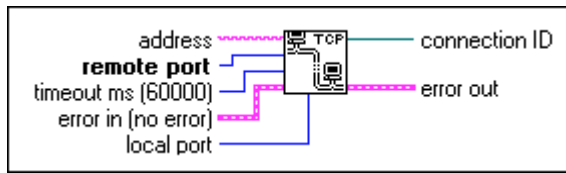
TCP Create Listener

Creates a listener for a TCP connection.



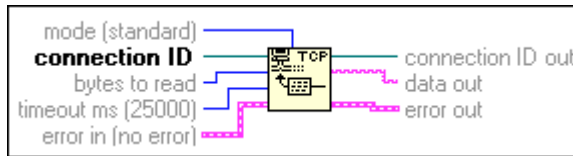
TCP Open Connection

Attempts to open a TCP connection with the specified address and port.



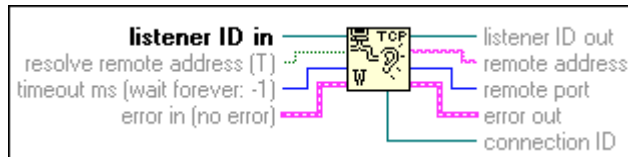
TCP Read

Receives up to **bytes to read** bytes from the specified TCP connection, returning the results in **data out**.



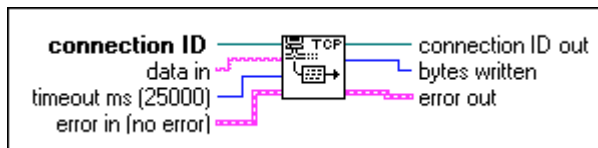
TCP Wait on Listener

Waits for an accepted TCP connection at the specified port.



TCP Write

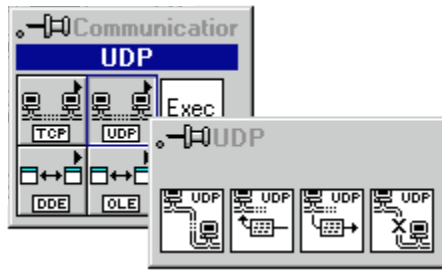
Writes the string **data in** to the specified TCP connection.



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** palette, which you access by selecting **Functions»Communication»UDP**.

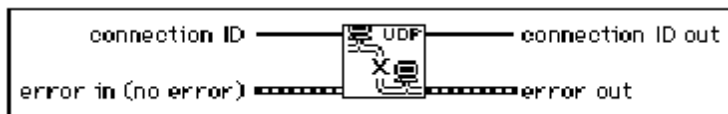


UDP VI Descriptions

The following UDP VIs are available.

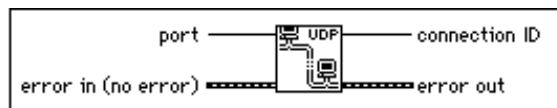
UDP Close

Closes the UDP connection specified by **connection ID**.



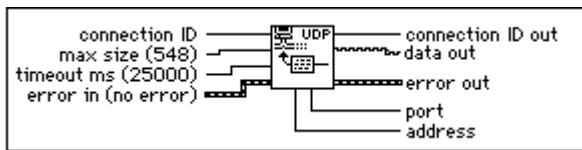
UDP Open

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.



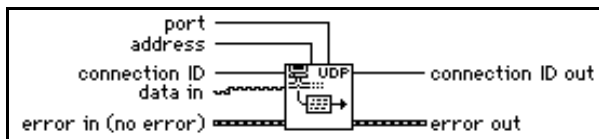
UDP Read

Returns a datagram in the string **data out** that has been received on the UDP connection specified by **connection ID**.



UDP Write

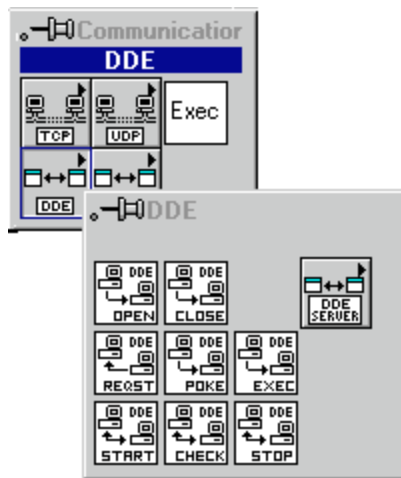
Writes the string **data in** to the remote UDP connection specified by **address** and **port**.



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** palette, which you access by selecting **Functions»Communication»DDE**.



The **DDE** palette includes the **DDE Server** subpalette.

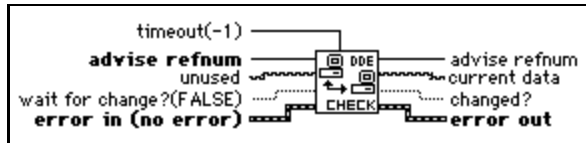
For examples of how to use the DDE VIs, see the examples in `examples\comm\DDEexamp.11b`.

DDE Client VI Descriptions

The following DDE Client VIs are available.

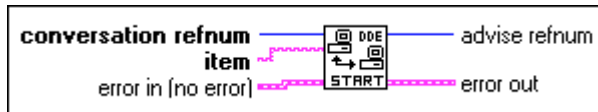
DDE Advise Check

Checks an advise value previously established by DDE Advise Start.



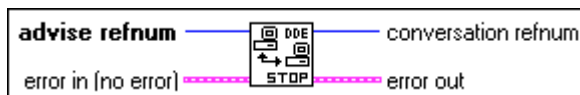
DDE Advise Start

Initiates an advise link.



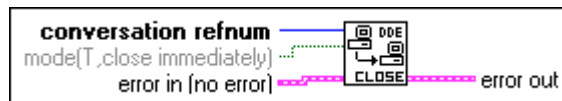
DDE Advise Stop

Cancels an advise link, previously established by DDE Advise Start.



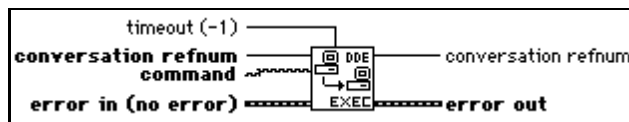
DDE Close Conversation

Closes a DDE conversation.



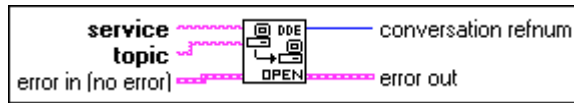
DDE Execute

Tells the DDE server to execute **command**.



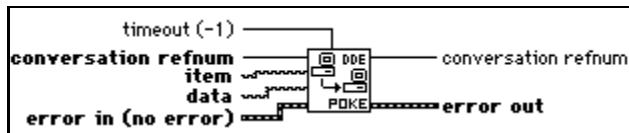
DDE Open Conversation

Establishes a connection between LabVIEW and another application. You must call this VI before you use any other DDE VIs (except Server VIs).



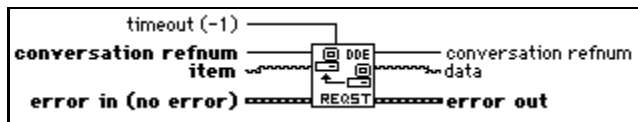
DDE Poke

Tells the DDE server to put the value **data** at **item**.



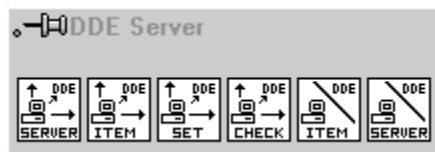
DDE Request

Initiates a DDE message exchange to obtain the current value of **item**.



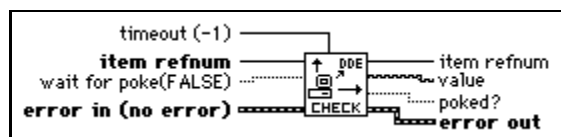
DDE Server VI Descriptions

You access the DDE Server functions by selecting **Functions»Communication»DDE»DDE Server**.



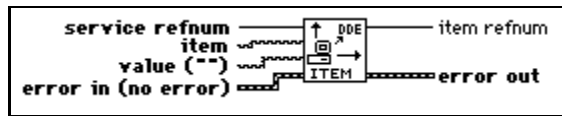
DDE Srv Check Item

Sets the value of a previously defined DDE Item.



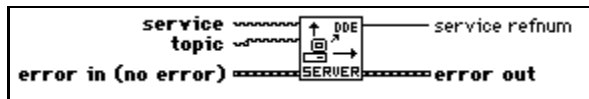
DDE Srv Register Item

Establishes a DDE item for the service specified by **service refnum**.



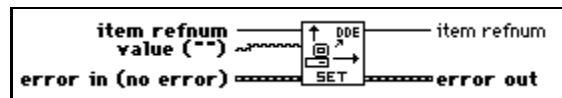
DDE Srv Register Service

Establishes a DDE service to which clients can connect.



DDE Srv Set Item

Sets the value of a previously defined DDE Item.

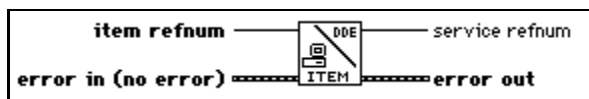


DDE Srv Unregister Item

Removes the specified item from its service.

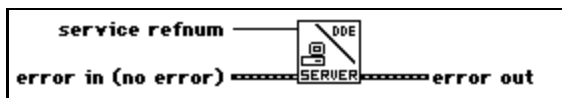


Note *DDE clients can no longer access the item after this VI completes.*



DDE Srv Unregister Service

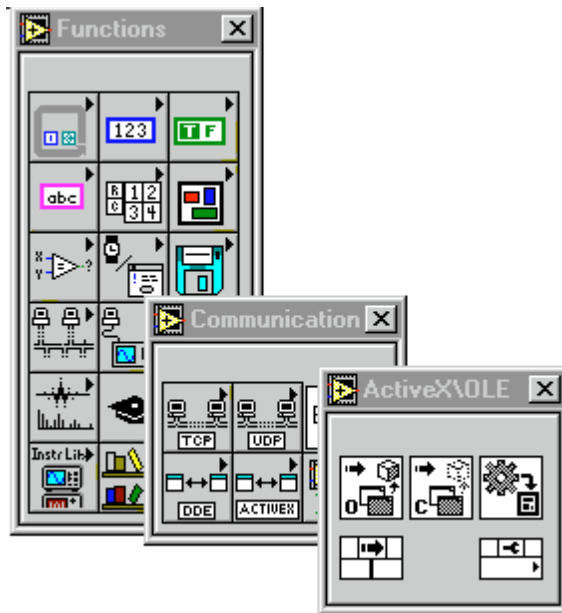
Removes the specified service. DDE clients can no longer connect to this service and all current conversations are closed.



ActiveX Automation Functions

This chapter describes the functions for support of ActiveX automation. These functions allow other ActiveX-enabled applications, such as Microsoft Excel, to request properties and methods from LabVIEW and individual VIs.

You access the ActiveX Automation functions by selecting **Functions»Communications»ActiveX/OLE**.



The **ActiveX/OLE** palette includes the following functions:

- Automation Open
- Automation Close
- Invoke Node
- Property Node

It also includes the ActiveX Variant to G Data function. For more information on this function, see *Data Conversion Function* later in this chapter.

National Instruments supports the old functions using compatibility functions, but all new applications should be built using the new functions. The following table shows how the old functions map to the new functions.

Table 51-1. New and Old ActiveX Automation Functions

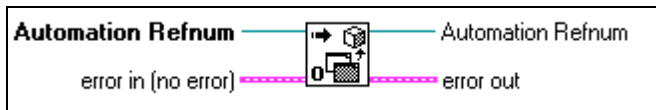
New ActiveX Functions	Old ActiveX Functions
Automation Open	Create Refnum
Automation Close	Release Refnum
Invoke Node	Execute Method
Property Node	Get Property Set Property

ActiveX Automation Function Descriptions

The following functions are available.

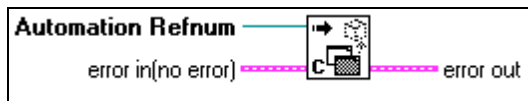
Open Automation Refnum

Opens an automation refnum which refers to a specific ActiveX Automation object. You select the class of the object by popping up on the function and selecting **Select ActiveX Class**. Once you open a refnum it can be passed to other ActiveX functions. You should select only createable classes as inputs to this function.



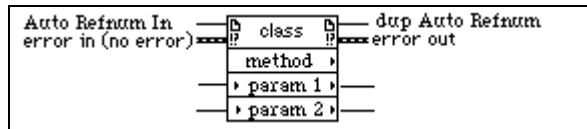
Close Automation Refnum

Closes an automation refnum. Make sure you close every open automation refnum when you no longer need it open.



Invoke Node

Invokes a method or action on an ActiveX object. To select an ActiveX class object, pop up and choose **Select»ActiveX Class** or wire an automation refnum to the input. To select a method related to that object, pop up on the second section of the node (“method” in the diagram) and select **Methods**. Once you select the method, the associated parameters appear below it. You can read or write to parameter values. Parameters with a white background are required inputs and the parameters with a gray background are optional inputs.

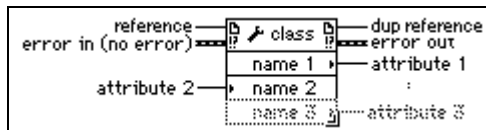


If the input parameters are of variant type, then you can wire in G data types and they will automatically be converted to variant data types and indicated by a coercion dot. If an output is of a variant type, use the ActiveX Variant to G function to convert to G type, if needed.

Property Node

Sets (writes) or gets (reads) ActiveX object property information. To select an ActiveX class object, pop up and choose **Select»ActiveX Class** or wire an automation refnum to the input. To select a property related to that object, pop up on the second line of the node and select **Properties**. To set property information, pop up and select **Change to Write**, and to get property information pop up and select **Change to Read**. Some properties are read or write only, so **Change to Write** or **Change to Read** respectively appears dimmed in the pop-up menu.

The Property Node works the same way as Attribute Nodes. If you want to add items to the node, pop up and select **Add Element** or click and drag the node to expand the number of items in the node. The properties are changed in the order from top to bottom. Remember if the small direction arrow on a property is on the left, you are setting the property value. If the small direction arrow on the property is on the right, you are getting the property value.



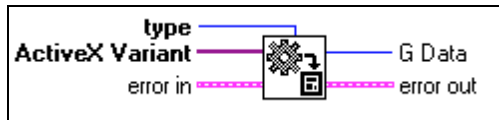
If the property to be written is of ActiveX Variant type, then you can wire in G data types and they will automatically be converted to variant data types and indicated by a coercion dot. If the property is of ActiveX Variant type, use the ActiveX Variant to G function to convert to G type, if needed.

Data Conversion Function

Some applications provide ActiveX data in the form of a self-describing data type called an *ActiveX* or *OLE Variant*. To review the data or process it in G, you must convert it to a corresponding V data type. To convert ActiveX Variant data to G data, use the ActiveX Variant to G Data function described below.

ActiveX Variant to G Data

Converts ActiveX Variant data to data that can be displayed in LabVIEW.



AppleEvent VIs

**Note**

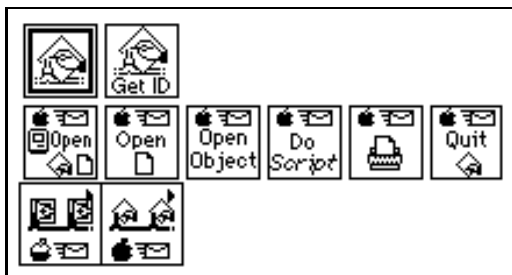
This chapter applies only to users running LabVIEW on the Macintosh System 7 platform.

This chapter describes the LabVIEW VIs for AppleEvents, one form of interapplication communication (IAC), through which Macintosh applications can communicate with each other. You also can use LabVIEW with a low-level form of IAC called 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 complex messages. This vocabulary is described in detail in the *AppleEvent Registry*, a document available from Apple Computer, Inc. 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 53, *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 7.0 or later with Program Linking enabled.

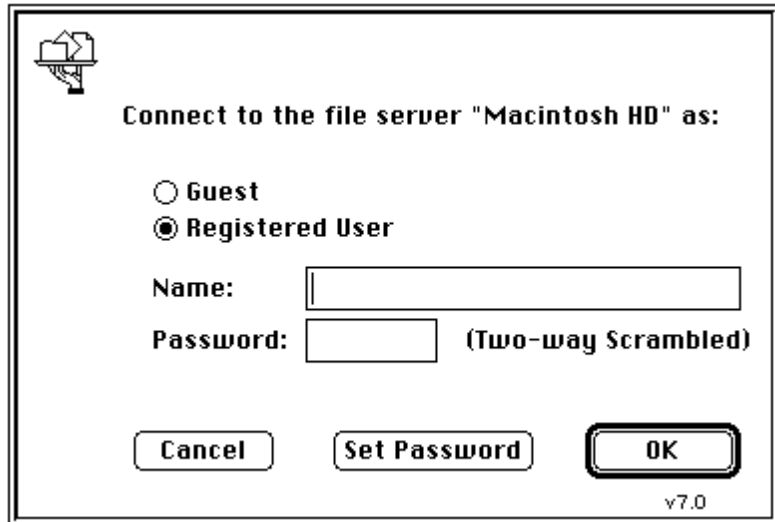
For examples of how to use the AppleEvent VIs, see the examples located in `examples:comm:AE Examples.11b`.

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.



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 might 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 receives 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.

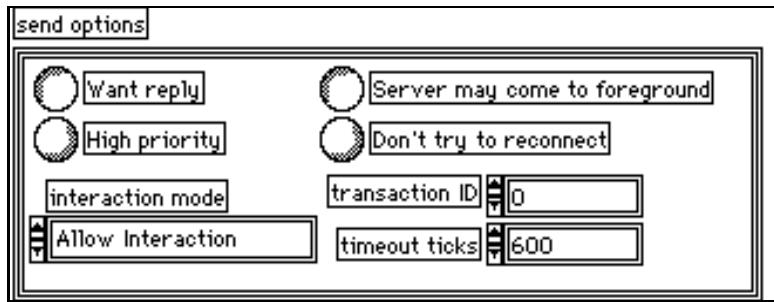
- PPC Browser creates the **target ID** by displaying a dialog box by which you interactively select AppleEvent-aware applications on the network.
- Get Target ID creates the **target ID** programmatically based on the application name and network location.

These VIs are discussed in more detail in the *Targeting VI Descriptions* 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.



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 name and location or the VI searches the entire network for the application.

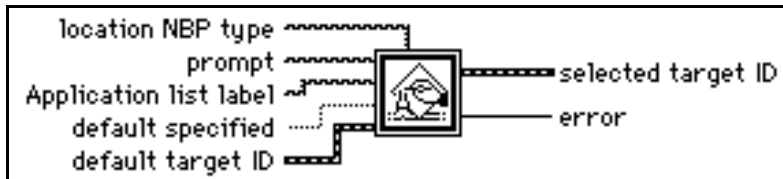


The following table summarizes the operation of **Search entire network**, **Zone**, and **Server**:

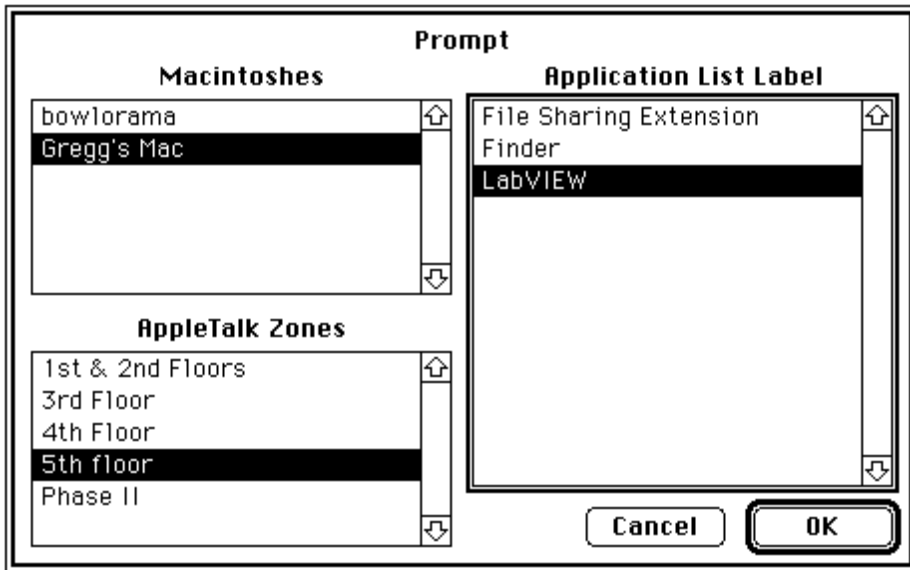
To search the following locations:	Use the following parameters:
The current computer	Zone and Server must be unwired. Search entire network must be FALSE.
A specific computer on the network	Zone and Server must specify the target computer's zone and server. (If you do not wire Zone , the VI searches 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.

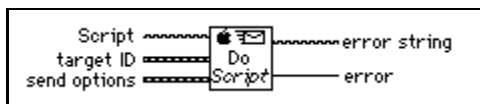


AppleEvent VI Descriptions

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.

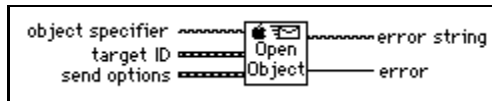


**Note**

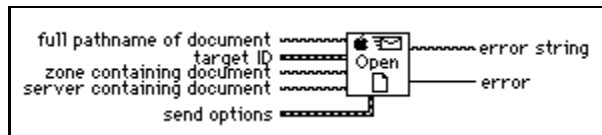
Apple may change the set of AppleEvents to which the Finder responds so that they more closely conform to the standard set of AppleEvents. As a result, the AppleEvent that AESend Finder Open sends to the Finder may not be supported in future versions of the system software.

AESend Open

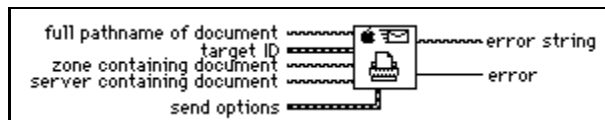
Sends the Open AppleEvent to a specified target application.

**AESend Open Document**

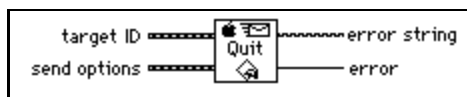
Sends the Open Document AppleEvent to the specified target application, telling the application to open the specified document.

**AESend Print Document**

Sends the Print Document AppleEvent to the specified target application, telling the application to print the specified document.

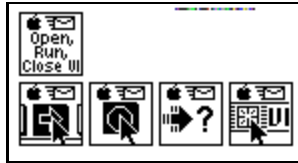
**AESend Quit Application**

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 Table A-5, *AppleEvent Error Codes*, of Appendix A, *Error Codes*, for error information.

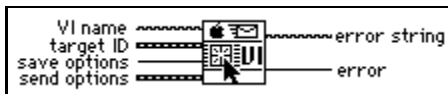
AESend Abort VI

Sends the Abort VI AppleEvent to the specified target LabVIEW application.



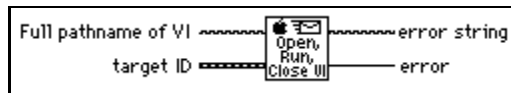
AESend Close VI

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 12, *Path and Refnum Controls and Indicators*, of your *G Programming Reference Manual* for a description of path controls and indicators available in the Controls palette.

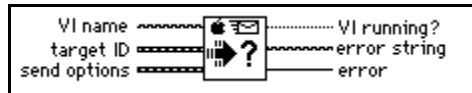
AESend Run VI

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 also can 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 `save`, 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 52-1 describes the format of AppleEvent descriptor strings and indicates VIs that can create the descriptor, where appropriate.

Table 52-1. 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 short	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
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 type code) 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]}	AECreat Record

Table 52-1. AppleEvent Descriptor String Formats (Continued)

To send data as:	Format the string as:	Parameter is of code type:	Examples:	VI that can construct string:
an AE descriptor list	Enclose a comma-separated list of descriptors in square brackets.	list	[123, -58, "test"]	AECreat 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)
some other data type	Embed data created in one of the types listed in 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 <code>errAECoeercionFail</code> error code if it cannot. If the data is of a different type, LabVIEW replaces the old type code with the specified type code.	The specified type code	sing(123 4) 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

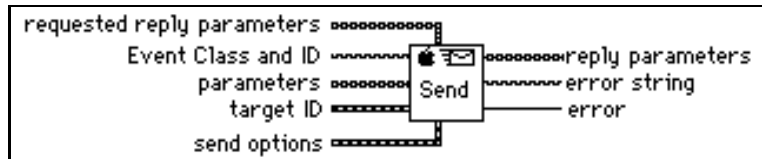
You can use Low-Level AppleEvent VIs to construct AppleEvent parameters and send the AppleEvent. The high-level VIs for sending AppleEvents, described earlier in this chapter, are based on the AESend VI, and are good examples of creating AppleEvents and their parameters.

To access the **Low Level Apple Events** palette, pop up on the **Low Level Apple Events** icon.



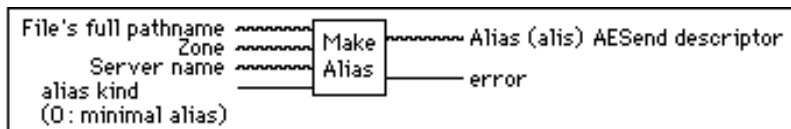
AESend

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.

AECreat 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 AECreat 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.

AECreat Logical Descriptor

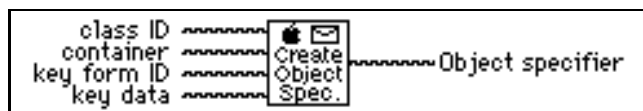
Creates a string describing an AppleEvent logical descriptor, which you use with the AESend VI.



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 AECreat Object Specifier VI to build a more complex descriptor string. See the *Object Support VI Example* section in this chapter for an example of its use.

AECreat Object Specifier

Creates a string describing an AppleEvent object, which you use with the AESend VI.



An object specifier is an AppleEvent record of type `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.

AECreat 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.

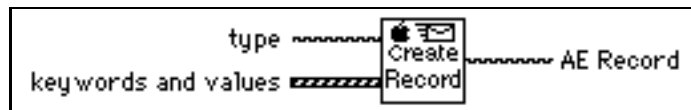
AECreat Descriptor List

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.



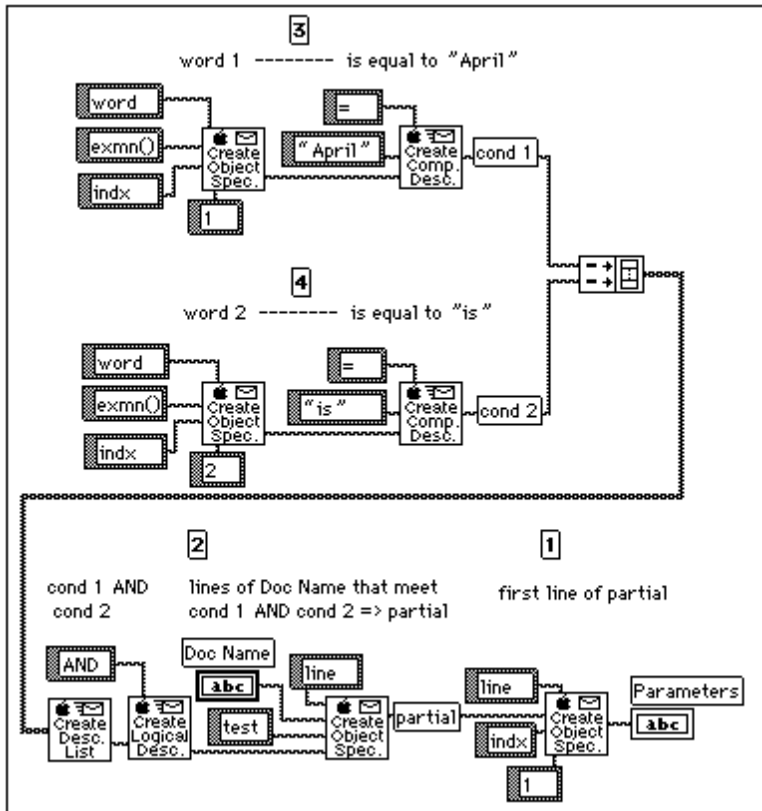
AECreat Record

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, instructing the word processor to return the first line of a specified document with the first word **April** and the second word **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:[
        compd{
          relo:=,
          obj1:"April",
          obj2:obj {
            want: type('word'),
            from: exmn( ),
            form: indx,
            seld: 1
          }
        },
        compd{
          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 24, *AppleEvents*, of the *LabVIEW User Manual*.

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 contains an error code. If the error is not a standard AppleEvent error, the reply also contains a string describing the error. Appendix A, *Error Codes*, 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 (sub VIs 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		

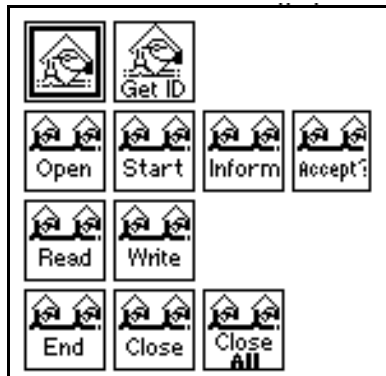
Possible Errors

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

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**.



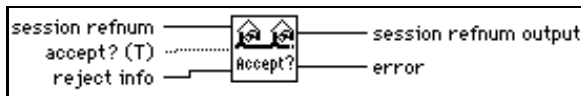
For examples of how to use the PPC VIs, see the examples located in `examples:comm:PPC Examples.llb`.

PPC VI Descriptions

The following PPC VIs are available.

PPC Accept Session

Accepts or rejects a PPC session request based on the Boolean **accept?**



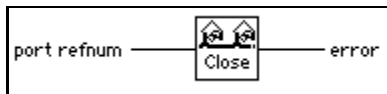
You should accept or reject the request using the PPC Accept Session VI immediately, because the other computer waits (hangs) until the VI accepts or rejects its attempt to initiate a session or an error occurs.

PPC Browser

For information on the PPC Browser VI, see Chapter 52, [AppleEvent VIs](#), of this manual.

Close All PPC Ports

Closes all the PPC ports that the PPC Open Port VI opened.

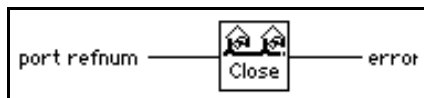


Closing a port terminates all outstanding calls associated with the port with a portClosedErr (error -916).

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

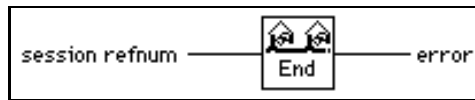
Closes the specified PPC port.



Closing a port terminates all outstanding calls associated with the port with a portClosedErr (error -916).

PPC End Session

Ends the specified PPC session.



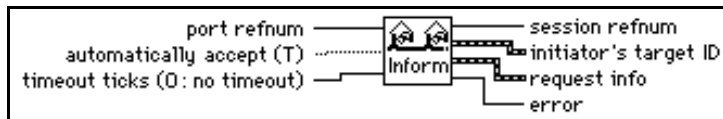
Ending a session causes all outstanding calls associated with the session (PPC Read and PPC Write calls) to finish with a sessClosedErr (error -917).

Get Target ID

For information on the Get Target ID VI, see Chapter 52, [AppleEvent VIs](#), of this manual.

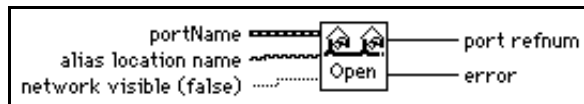
PPC Inform Session

Waits for a PPC session request.

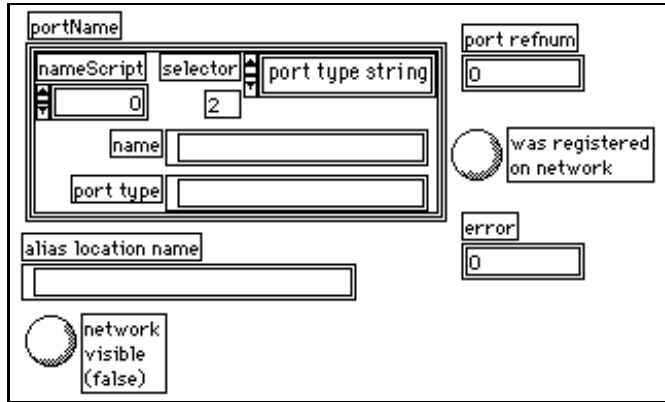


PPC Open Port

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.



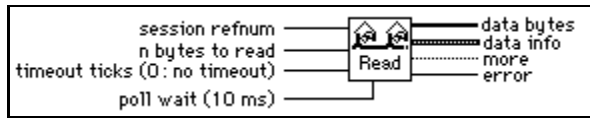
When opening a port using PPC Open Port, you must specify a **portName** cluster.



Refer to the LabVIEW online help for more information on this VI.

PPC Read

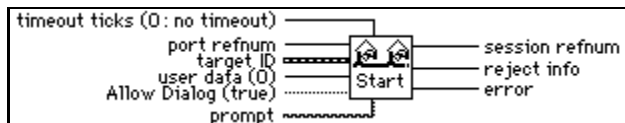
Reads a block of information from a specified session. If a timeout occurs or the VI aborts before completing execution, the port that **port refnum** represents closes.



PPC Read executes asynchronously by starting to read the specified data and then polling until the read is finished.

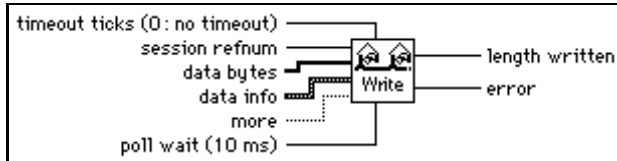
PPC Start Session

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.



Error Codes

This document contains tables listing all the numeric error codes for LabVIEW.

Connect error handler VIs to other VIs to return a description of an error, if one occurs. Error handler VIs also can display a dialog box with an error message description and with buttons that can stop or continue execution. See the *Error Handling* topic in the *LabVIEW Online Reference* for more information about error handlers.



Note

All error codes and descriptions are also included in the configuration utility help panels in Windows and Macintosh platforms.

Numeric Error Codes

The tables are arranged roughly in ascending order, from negative to positive values. Tables with negative number values are arranged from the smallest absolute value to the largest absolute value. Notice that error codes 5000 to 9999 are reserved for your own use.

Table A-1. Numeric Error Code Ranges

Error Code Range				Table
*	-1073807360	to	-1073807231	VISA Error Codes
	-20001	to	-20065	Analysis VI Error Codes
	-10001	to	-10943	Data Acquisition VI Error Codes
	-1700	to	-1719	AppleEvent Error Codes
*	-1200	to	-13xx	Instrument Driver Error Codes
	-900	to	-932	PPC Error Codes
*	0	to	85	LabVIEW Function Error Codes
*	0	to	32	GPIB Error Codes
	1	to	5	LabVIEW Specific PPC Error Codes
*	53	to	66	TCP/IP and UDP Error Codes
*	61	to	65	Serial Port Error Codes

Table A-1. Numeric Error Code Ranges (Continued)

Error Code Range	Table
1000 to 1004	LabVIEW Specific AppleEvent Error Codes
14001 to 14020	DDE Error Codes

* These tables contain some error codes with overlapping numerical values but different meanings, depending on the source of the error.

Table A-2. VISA Error Codes

Error Code	Error Name	Description
-1073807360	VI_ERROR_SYSTEM_ERROR	Unknown system error (miscellaneous error).
-1073807346	VI_ERROR_INV_OBJECT VI_ERROR_INV_SESSION	The given session or object reference is invalid.
-1073807345	VI_ERROR_RSRC_LOCKED	Specified type of lock cannot be obtained or specified operation cannot be performed because the resource is locked.
-1073807344	VI_ERROR_INV_EXPR	Invalid expression specified for search.
-1073807343	VI_ERROR_RSRC_NFOUND	Insufficient location information, or the device or resource is not present in the system.
-1073807342	VI_ERROR_INV_RSRC_NAME	Invalid resource reference specified. Parsing error.
-1073807341	VI_ERROR_INV_ACC_MODE	Invalid access mode.
-1073807339	VI_ERROR_TMO	Timeout expired before operation completed.
-1073807338	VI_ERROR_CLOSING_FAILED	Unable to deallocate the previously allocated data structures corresponding to this session or object reference.
-1073807332	VI_ERROR_INV_JOB_ID	Specified job identifier is invalid.
-1073807331	VI_ERROR_NSUP_ATTR	The specified attribute is not defined or supported by the referenced resource.
-1073807330	VI_ERROR_NSUP_ATTR_STATE	The specified state of the attribute is not valid, or is not supported as defined by the resource.
-1073807329	VI_ERROR_ATTR_READONLY	The specified attribute is read-only.
-1073807322	VI_ERROR_INV_EVENT	Specified event type is not supported by the resource.
-1073807321	VI_ERROR_INV_MECH	Invalid mechanism specified.
-1073807320	VI_ERROR_HNDLR_NINSTALLED	A handler was not installed.
-1073807319	VI_ERROR_INV_HNDLR_REF	The given handler reference is invalid.
-1073807318	VI_ERROR_INV_CONTEXT	Specified event context is invalid.

Table A-2. VISA Error Codes (Continued)

Error Code	Error Name	Description
-1073807308	VI_ERROR_RAW_WR_PROT_VIOL	Violation of raw write protocol occurred during transfer.
-1073807307	VI_ERROR_RAW_RD_PROT_VIOL	Violation of raw read protocol occurred during transfer.
-1073807306	VI_ERROR_OUTP_PROT_VIOL	Device reported an output protocol error during transfer.
-1073807305	VI_ERROR_INP_PROT_VIOL	Device reported an input protocol error during transfer.
-1073807304	VI_ERROR_BERR	Bus error occurred during transfer.
-1073807302	VI_ERROR_INV_SETUP	Unable to start operation because setup is invalid (due to attributes being set to an inconsistent state).
-1073807300	VI_ERROR_ALLOC	Insufficient system resources to perform necessary memory allocation.
-1073807299	VI_ERROR_INV_MASK	Invalid buffer mask specified.
-1073807298	VI_ERROR_IO	Could not perform read/write operation because of I/O error.
-1073807297	VI_ERROR_INV_FMT	A format specifier in the format string is invalid.
-1073807295	VI_ERROR_NSUP_FMT	A format specifier in the format string is not supported.
-1073807294	VI_ERROR_LINE_IN_USE	The specified trigger line is currently in use.
-1073807286	VI_ERROR_SRQ_NOCCURRED	Service request has not been received for the session.
-1073807282	VI_ERROR_INV_SPACE	Invalid address space specified.
-1073807279	VI_ERROR_INV_OFFSET	Invalid offset specified.
-1073807276	VI_ERROR_NSUP_OFFSET	Specified offset is not accessible from this hardware.
-1073807273	VI_ERROR_WINDOW_NMAPPED	The specified session is not currently mapped.
-1073807265	VI_ERROR_NLISTENERS	No Listeners condition is detected (both NRFD and NDAC are deasserted).
-1073807264	VI_ERROR_NCIC	The interface associated with this session is not currently the controller in charge.
-1073807257	VI_ERROR_NSUP_OPER	The given session or object reference does not support this operation.
-1073807242	VI_ERROR_NSUP_WIDTH	Specified width is not supported by this hardware.
-1073807239	VI_ERROR_INV_PROT	The protocol specified is invalid.
-1073807237	VI_ERROR_INV_SIZE	Invalid size of window specified.
-1073807232	VI_ERROR_WINDOW_MAPPED	The specified session already contains a mapped window.
-1073807231	VI_ERROR_NIMPL_OPER	The given operation is not implemented.

Table A-3. Analysis Error Codes

Error Code	Error Name	Description
0	NoErr	No error; the call was successful.
-20001	OutOfMemErr	There is not enough memory left to perform the specified routine.
-20002	EqSamplesErr	The input sequences must be the same size.
-20003	SamplesGTZeroErr	The number of samples must be greater than zero.
-20004	SamplesGEZeroErr	The number of samples must be greater than or equal to zero.
-20005	SamplesGEOneErr	The number of samples must be greater than or equal to one.
-20006	SamplesGETwoErr	The number of samples must be greater than or equal to two.
-20007	SamplesGEThreeErr	The number of samples must be greater than or equal to three.
-20008	ArraySizeErr	The input arrays do not contain the correct number of data values for this VI.
-20009	PowerOfTwoErr	The size of the input array must be a power of two: size = 2^m , $0 < m < 23$.
-20010	MaxXformSizeErr	The maximum transform size has been exceeded.
-20011	DutyCycleErr	The duty cycle must meet the condition: $0 \leq \text{duty cycle} \leq 100$.
-20012	CyclesErr	The number of cycles must be greater than zero and less than or equal to the number of samples.
-20013	WidthLTSamplesErr	The width must meet the condition: $0 < \text{width} < \text{samples}$.
-20014	DelayWidthErr	The delay must meet the condition: $0 \leq (\text{delay} + \text{width}) < \text{samples}$.
-20015	DtGEZeroErr	dt must be greater than or equal to zero.
-20016	DtGTZeroErr	dt must be greater than zero.
-20017	IndexLTSamplesErr	The index must meet the condition: $0 \leq \text{index} < \text{samples}$.
-20018	IndexLengthErr	The index must meet the condition: $0 \leq (\text{index} + \text{length}) < \text{samples}$.
-20019	UpperGELowerErr	The upper value must be greater than or equal to the lower value.

Table A-3. Analysis Error Codes (Continued)

Error Code	Error Name	Description
-20020	NyquistErr	The cutoff frequency, f_c , must meet the condition: $0 \leq f_c \leq \frac{f_s}{2}$.
-20021	OrderGTZeroErr	The order must be greater than zero.
-20022	DecFactErr	The decimating factor must meet the condition: $0 < \text{decimating} \leq \text{samples}$.
-20023	BandSpecErr	The band specifications must meet the condition: $0 \leq f_{low} \leq f_{high} \leq \frac{f_s}{2}$.
-20024	RippleGTZeroErr	The ripple amplitude must be greater than zero.
-20025	AttenGTZeroErr	The attenuation must be greater than zero.
-20026	WidthGTZeroErr	The width must be greater than zero.
-20027	FinalGTZeroErr	The final value must be greater than zero.
-20028	AttenGTRippleErr	The attenuation must be greater than the ripple amplitude.
-20029	StepSizeErr	The step-size, μ , must meet the condition: $0 \leq \mu \leq 0.1$.
-20030	LeakErr	The leakage coefficient must meet the condition: $0 \leq \text{leak} \leq \mu$.
-20031	EqRplDesignErr	The filter cannot be designed with the specified input values.
-20032	RankErr	The rank of the filter must meet the condition: $1 \leq (2 \times \text{rank} + 1) \leq \text{size}$.
-20033	EvenSizeErr	The number of coefficients must be odd for this filter.
-20034	OddSizeErr	The number of coefficients must be even for this filter.
-20035	StdDevErr	The standard deviation must be greater than zero for normalization.
-20036	MixedSignErr	The elements of the Y Values array must be nonzero and either all positive or all negative.
-20037	SizeGTOrderErr	The number of data points in the Y Values array must be greater than two.
-20038	IntervalsErr	The number of intervals must be greater than zero.
-20039	MatrixMulErr	The number of columns in the first matrix is not equal to the number of rows in the second matrix or vector.
-20040	SquareMatrixErr	The input matrix must be a square matrix.

Table A-3. Analysis Error Codes (Continued)

Error Code	Error Name	Description
-20041	SingularMatrixErr	The system of equations cannot be solved because the input matrix is singular.
-20042	LevelsErr	The number of levels is out of range.
-20043	FactorErr	The level of factors is out of range for some data.
-20044	ObservationsErr	Zero observations were made at some level of a factor.
-20045	DataErr	The total number of data points must be equal to the product of the levels for each factor and the observations per cell.
-20046	OverflowErr	There is an overflow in the calculated F-value.
-20047	BalanceErr	The data is unbalanced. All cells must contain the same number of observations.
-20048	ModelErr	The Random Effect model was requested when the Fixed Effect model was required.
-20049	DistinctErr	The x values must be distinct.
-20050	PoleErr	The interpolating function has a pole at the requested value.
-20051	ColumnErr	All values in the first column in the X matrix must be 1.
-20052	FreedomErr	The degrees of freedom must be one or more.
-20053	ProbabilityErr	The probability must be between zero and one.
-20054	InvProbErr	The probability must be greater than or equal to zero and less than one.
-20055	CategoryErr	The number of categories or samples must be greater than one.
-20056	TableErr	The contingency table must not contain a negative number.
-20061	InvSelectionErr	One of the input selections is invalid.
-20062	MaxIterErr	The maximum iterations have been exceeded.
-20063	PolyErr	The polynomial coefficients are invalid.
-20064	InitStateErr	This VI has not been initialized correctly.
-20065	ZeroVectorErr	The vector cannot be zero.

Table A-4. Data Acquisition VI Error Codes

Error Code	Error Name	Description
-10001	syntaxError	An error was detected in the input string; the arrangement or ordering of the characters in the string is not consistent with the expected ordering.
-10002	semanticsError	An error was detected in the input string; the syntax of the string is correct, but certain values specified in the string are inconsistent with other values specified in the string.
-10003	invalidValueError	The value of a numeric parameter is invalid.
-10004	valueConflictError	The value of a numeric parameter is inconsistent with another parameter, and the combination is therefore invalid.
-10005	badDeviceError	The device parameter is invalid.
-10006	badLineError	The line parameter is invalid.
-10007	badChanError	A channel is out of range for the board type or input configuration, the combination of channels is not allowed, or you must reverse the scan order so that channel 0 is last.
-10008	badGroupError	The group is invalid.
-10009	badCounterError	The counter is invalid.
-10010	badCountError	The count is too small or too large for the specified counter; or the given I/O transfer count is not appropriate for the current buffer or channel configuration.
-10011	badIntervalError	The analog input scan rate is too fast for the number of channels and the channel clock rate; or the given clock rate is not supported by the associated counter channel or I/O channel.
-10012	badRangeError	The analog input or analog output voltage range is invalid for the specified channel.
-10013	badErrorCodeError	The driver returned an unrecognized or unlisted error code.
-10014	groupTooLargeError	The group size is too large for the board.
-10015	badTimeLimitError	The time limit is invalid.
-10016	badReadCountError	The read count is invalid.
-10017	badReadModeError	The read mode is invalid.
-10018	badReadOffsetError	The offset is unreachable.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10019	badClkFrequencyError	The frequency is invalid.
-10020	badTimebaseError	The timebase is invalid.
-10021	badLimitsError	The limits are beyond the range of the board.
-10022	badWriteCountError	Your data array contains an incomplete update, or you are trying to write past the end of the internal buffer, or your output operation is continuous and the length of your array is not a multiple of one half of the internal buffer size.
-10023	badWriteModeError	The write mode is out of range or is disallowed.
-10024	badWriteOffsetError	Adding the write offset to the write mark places the write mark outside the internal buffer.
-10025	limitsOutOfRangeError	The requested input limits exceed the board's capability or configuration. Alternate limits were selected.
-10026	badBufferSpecificationError	The requested number of buffers or the buffer size is not allowed; for example, Lab-PC buffer limit is 64K samples, or the board does not support multiple buffers.
-10027	badDAQEventError	For DAQEvents 0 and 1, general value A must be greater than 0 and less than the internal buffer size. If DMA is used for DAQEvent 1, general value A must divide the internal buffer size evenly. If the TIO-10 is used for DAQEvent 4, general value A must be 1 or 2.
-10028	badFilterCutoffError	The cutoff frequency specified is not valid for this device.
-10029	obsoleteFunctionError	The function you are calling is no longer supported in this version of the driver.
-10030	badBaudRateError	The specified baud rate for communicating with the serial port is not valid on this platform.
-10031	badChassisIDError	The specified SCXI chassis does not correspond to a configured SCXI chassis.
-10032	badModuleSlotError	The SCXI module slot that was specified is invalid or corresponds to an empty slot.
-10033	invalidWinHandleError	The window handle passed to the function is invalid.
-10034	noSuchMessageError	No configured message matches the one you tried to delete.
-10080	badGainError	The gain is invalid.
-10081	badPretrigCountError	The pretrigger sample count is invalid.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10082	badPosttrigCountError	The posttrigger sample count is invalid.
-10083	badTrigModeError	The trigger mode is invalid.
-10084	badTrigCountError	The trigger count is invalid.
-10085	badTrigRangeError	The trigger range or trigger hysteresis window is invalid.
-10086	badExtRefError	The external reference value is invalid.
-10087	badTrigTypeError	The trigger type parameter is invalid.
-10088	badTrigLevelError	The trigger level is invalid.
-10089	badTotalCountError	The total count is inconsistent with the buffer size and pretrigger scan count or with the board type.
-10090	badRPGError	The individual range, polarity, and gain settings are valid but the combination specified is not allowed.
-10091	badIterationsError	You have attempted to use an invalid setting for the iterations parameter. The iterations value must be 0 or greater. Your device might be limited to only two values, 0 and 1.
-10092	lowScanIntervalError	Some devices require a time gap between the last sample in a scan and the start of the next scan. The scan interval you have specified does not provide a large enough gap for the board. See the <code>SCAN_start</code> function in the language interface API for an explanation.
-10093	fifoModeError	FIFO mode waveform generation cannot be used because at least one condition is not satisfied.
-10100	badPortWidthError	The requested digital port width is not a multiple of the hardware port width or is not attainable by the DAQ hardware.
-10120	gpctrBadApplicationError	Invalid application used.
-10121	gpctrBadCtrNumberError	Invalid counterNumber used.
-10122	gpctrBadParamValueError	Invalid paramValue used.
-10123	gpctrBadParamIDError	Invalid paramID used.
-10124	gpctrBadEntityIDError	Invalid entityID used.
-10125	gpctrBadActionError	Invalid action used.
-10200	EEPROMreadError	Unable to read data from EEPROM.
-10201	EEPROMwriteError	Unable to write data to EEPROM.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10240	noDriverError	The driver interface could not locate or open the driver.
-10241	oldDriverError	One of the driver files or the configuration utility is out of date.
-10242	functionNotFoundError	The specified function is not located in the driver.
-10244	deviceInitError	The driver encountered a hardware-initialization error while attempting to configure the specified device.
-10245	osInitError	The driver encountered an operating system error while attempting to perform an operation, or the operating system does not support an operation performed by the driver.
-10246	communicationsError	The driver is unable to communicate with the specified external device.
-10248	dupAddressError	The base addresses for two or more devices are the same; consequently, the driver is unable to access the specified device.
-10249	intConfigError	The interrupt configuration is incorrect given the capabilities of the computer or device.
-10250	dupIntError	The interrupt levels for two or more devices are the same.
-10251	dmaConfigError	The DMA configuration is incorrect given the capabilities of the computer/DMA controller or device.
-10252	dupDMAError	The DMA channels for two or more devices are the same.
-10253	jumperlessBoardError	Unable to find one or more jumperless boards you have configured using the NI-DAQ Configuration Utility.
-10254	DAQCardConfError	Cannot configure the DAQCard because of one of the following reasons: <ol style="list-style-type: none"> 1. The correct version of the card and socket services software is not installed. 2. The card in the PCMCIA socket is not a DAQCard. 3. The base address and/or interrupt level requested are not available according to the card and socket services resource manager. 4. The Card Services failed to load due to insufficient available memory under 1 MB in Windows 3.1. Try different settings or use AutoAssign in the NI-DAQ Configuration Utility. Memory under 1 MB must be available to configure DAQCard in Win 3.x.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10255	remoteChassisDriverInitError	There was an error in initializing the driver for remote SCXI.
-10256	comPortOpenError	There was an error in opening the specified COM port.
-10257	baseAddressError	Bad base address specified in the configuration utility.
-10258	dmaChannel1Error	Bad DMA channel 1 specified in the configuration utility or by the operating system.
-10259	dmaChannel2Error	Bad DMA channel 2 specified in the configuration utility or by the operating system.
-10260	dmaChannel3Error	Bad DMA channel 3 specified in the configuration utility or by the operating system.
-10261	userModeToKernelModeCallError	The user mode code failed when calling the kernel mode.
-10340	noConnectError	No RTSI signal/line is connected, or the specified signal and the specified line are not connected.
-10341	badConnectError	The RTSI signal/line cannot be connected as specified.
-10342	multConnectError	The specified RTSI signal is already being driven by an RTSI line, or the specified RTSI line is already being driven by an RTSI signal.
-10343	SCXIConfigError	The specified SCXI configuration parameters are invalid, or the function cannot be executed given the current SCXI configuration.
-10344	chassisSynchedError	The Remote SCXI unit is not synchronized with the host. Reset the chassis again to resynchronize it with the host.
-10345	chassisMemAllocError	The required amount of memory cannot be allocated on the Remote SCXI unit for the specified operation.
-10346	badPacketError	The packet received by the Remote SCXI unit is invalid. Check your serial port cable connections.
-10347	chassisCommunicationError	There was an error in sending a packet to the remote chassis. Check your serial port cable connections.
-10348	waitingForReprogError	The remote SCXI unit is in reprogramming mode and is waiting for reprogramming commands from the host (NI-DAQ Configuration Utility).
-10349	SCXIModuleTypeConflictError	The module ID read from the SCXI module conflicts with the configured module type.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10370	badScanListError	The scan list is invalid; for example, you are mixing AMUX-64T channels and onboard channels, scanning SCXI channels out of order, or have specified a different starting channel for the same SCXI module. Also, the driver attempts to achieve complicated gain distributions over SCXI channels on the same module by manipulating the scan list and returns this error if it fails.
-10400	userOwnedRsrcError	The specified resource is owned by the user and cannot be accessed or modified by the driver.
-10401	unknownDeviceError	The specified device is not a National Instruments product, or the driver does not support the device (for example, the driver was released before the device was supported).
-10402	deviceNotFoundError	No device is located in the specified slot or at the specified address.
-10404	noLineAvailError	No line is available.
-10405	noChanAvailError	No channel is available.
-10406	noGroupAvailError	No group is available.
-10407	lineBusyError	The specified line is in use.
-10408	chanBusyError	The specified channel is in use.
-10409	groupBusyError	The specified group is in use.
-10410	relatedLCGBusyError	A related line, channel, or group is in use; if the driver configures the specified line, channel, or group, the configuration, data, or handshaking lines for the related line, channel, or group will be disturbed.
-10411	counterBusyError	The specified counter is in use.
-10412	noGroupAssignError	No group is assigned, or the specified line or channel cannot be assigned to a group.
-10413	groupAssignError	A group is already assigned, or the specified line or channel is already assigned to a group.
-10414	reservedPinError	The selected signal requires a pin that is reserved and configured only by NI-DAQ. You cannot configure this pin yourself.
-10415	externalMuxSupportError	This function does not support this device when an external multiplexer (such as an AMUX-64T or SCXI) is connected to it.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10440	sysOwnedRsrcError	The specified resource is owned by the driver and cannot be accessed or modified by the user.
-10441	memConfigError	No memory is configured to support the current data-transfer mode, or the configured memory does not support the current data-transfer mode. (If block transfers are in use, the memory must be capable of performing block transfers.)
-10442	memDisabledError	The specified memory is disabled or is unavailable given the current addressing mode.
-10443	memAlignmentError	The transfer buffer is not aligned properly for the current data-transfer mode. For example, the buffer is at an odd address, is not aligned to a 32-bit boundary, is not aligned to a 512-bit boundary, and so on. Alternatively, the driver is unable to align the buffer because the buffer is too small.
-10445	memLockError	The transfer buffer cannot be locked into physical memory. On PC AT machines, portions of the DMA data acquisition buffer may be in an invalid DMA region, for example, above 16 MB.
-10446	memPageError	The transfer buffer contains a page break; system resources may require reprogramming when the page break is encountered.
-10447	memPageLockError	The operating environment is unable to grant a page lock.
-10448	stackMemError	The driver is unable to continue parsing a string input due to stack limitations.
-10449	cacheMemError	A cache-related error occurred, or caching is not supported in the current mode.
-10450	physicalMemError	A hardware error occurred in physical memory, or no memory is located at the specified address.
-10451	virtualMemError	The driver is unable to make the transfer buffer contiguous in virtual memory and therefore cannot lock the buffer into physical memory; thus, you cannot use the buffer for DMA transfers.
-10452	noIntAvailError	No interrupt level is available for use.
-10453	intInUseError	The specified interrupt level is already in use by another device.
-10454	noDMAError	No DMA controller is available in the system.
-10455	noDMAAvailError	No DMA channel is available for use.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10456	DMAInUseError	The specified DMA channel is already in use by another device.
-10457	badDMAGroupError	DMA cannot be configured for the specified group because it is too small, too large, or misaligned. Consult the user manual for the device in question to determine group ramifications with respect to DMA.
-10458	diskFullError	A disk overflow occurred while attempting to write to a file.
-10459	DLLInterfaceError	The DLL could not be called because of an interface error.
-10460	interfaceInteractionError	You have mixed VIs from the DAQ library and the _DAQ compatibility library (LabVIEW 2.2 VIs). You can switch between the two libraries only by running the DAQ VI Device Reset before calling _DAQ compatibility VIs or by running the compatibility VI Board Reset before calling DAQ VIs.
-10480	muxMemFullError	The scan list is too large to fit into the mux-gain memory of the board.
-10481	bufferNotInterleavedError	You must provide a single buffer of interleaved data, and the channels must be in ascending order. You cannot use DMA to transfer data from two buffers; however, you may be able to use interrupts.
-10540	SCXIModuleNotSupportedError	At least one of the SCXI modules specified is not supported for the operation.
-10541	TRIG1ResourceConflict	CTRB1 will drive COUTB1. However, CTRB1 also will drive TRIG1. This conflict might cause unpredictable results when the chassis is scanned.
-10600	noSetupError	No setup operation has been performed for the specified resources. Or, some resources require a specific ordering of calls for proper setup.
-10601	multSetupError	The specified resources have already been configured by a setup operation.
-10602	noWriteError	No output data has been written into the transfer buffer.
-10603	groupWriteError	The output data associated with a group must be for a single channel or for consecutive channels.
-10604	activeWriteError	Once data generation has started, only the transfer buffers originally written to can be updated. If DMA is active and a single transfer buffer contains interleaved channel-data, new data must be provided for all output channels currently using the DMA channel.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10605	endWriteError	No data was written to the transfer buffer because the final data block has already been loaded.
-10606	notArmedError	The specified resource is not armed.
-10607	armedError	The specified resource is already armed.
-10608	noTransferInProgError	No transfer is in progress for the specified resource.
-10609	transferInProgError	A transfer is already in progress for the specified resource, or the operation is not allowed because the device is in the process of performing transfers, possibly with different resources.
-10610	transferPauseError	A single output channel in a group cannot be paused if the output data for the group is interleaved.
-10611	badDirOnSomeLinesError	Some of the lines in the specified channel are not configured for the transfer direction specified. For a write transfer, some lines were configured for input. For a read transfer, some lines were configured for output.
-10612	badLineDirError	The specified line does not support the specified transfer direction.
-10613	badChanDirError	The specified channel does not support the specified transfer direction.
-10614	badGroupDirError	The specified group does not support the specified transfer direction.
-10615	masterClkError	The clock configuration for the clock master is invalid.
-10616	slaveClkError	The clock configuration for the clock slave is invalid.
-10617	noClkSrcError	No source signal has been assigned to the clock resource.
-10618	badClkSrcError	The specified source signal cannot be assigned to the clock resource.
-10619	multClkSrcError	A source signal has already been assigned to the clock resource.
-10620	noTrigError	No trigger signal has been assigned to the trigger resource.
-10621	badTrigError	The specified trigger signal cannot be assigned to the trigger resource.
-10622	preTrigError	The pretrigger mode is not supported or is not available in the current configuration, or no pretrigger source has been assigned.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10623	postTrigError	No posttrigger source has been assigned.
-10624	delayTrigError	The delayed trigger mode is not supported or is not available in the current configuration, or no delay source has been assigned.
-10625	masterTrigError	The trigger configuration for the trigger master is invalid.
-10626	slaveTrigError	The trigger configuration for the trigger slave is invalid.
-10627	noTrigDrvError	No signal has been assigned to the trigger resource.
-10628	multTrigDrvError	A signal has already been assigned to the trigger resource.
-10629	invalidOpModeError	The specified operating mode is invalid, or the resources have not been configured for the specified operating mode.
-10630	invalidReadError	The parameters specified to read data were invalid in the context of the acquisition. For example, an attempt was made to read 0 bytes from the transfer buffer, or an attempt was made to read past the end of the transfer buffer.
-10631	noInfiniteModeError	Continuous input or output transfers are not allowed in the current operating mode.
-10632	someInputsIgnoredError	Certain inputs were ignored because they are not relevant in the current operating mode.
-10633	invalidRegenModeError	The specified analog output regeneration mode is not allowed for this board.
-10634	noContTransferInProgressError	No continuous (double-buffered) transfer is in progress for the specified resource.
-10635	invalidSCXIOPModeError	Either the SCXI operating mode specified in a configuration call is invalid, or a module is in the wrong operating mode to execute the function call.
-10636	noContWithSynchError	You cannot start a continuous (double-buffered) operation with a synchronous function call.
-10637	bufferAlreadyConfigError	Attempted to configure a buffer after the buffer had already been configured. You can configure a buffer only once.
-10680	badChanGainError	All channels of this board must have the same gain.
-10681	badChanRangeError	All channels of this board must have the same range.
-10682	badChanPolarityError	All channels of this board must have the same polarity.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10683	badChanCouplingError	All channels of this board must have the same coupling.
-10684	badChanInputModeError	All channels of this board must have the same input mode.
-10685	clkExceedsBrdsMaxConvRateError	The clock rate selected exceeds the recommended maximum rate for this board.
-10686	scanListInvalidError	A configuration change has invalidated the scan list.
-10687	bufferInvalidError	A configuration change has invalidated the acquisition buffer, or an acquisition buffer has not been configured.
-10688	noTrigEnabledError	The total number of scans and pretrigger scans implies that a trigger start is intended, but no trigger is enabled.
-10689	digitalTrigBError	Digital trigger B is illegal for the total scans and pretrigger scans specified.
-10690	digitalTrigAandBError	This board does not allow digital triggers A and B to be enabled at the same time.
-10691	extConvRestrictionError	This board does not allow an external sample clock with an external scan clock, start trigger, or stop trigger.
-10692	chanClockDisabledError	Cannot start the acquisition because the channel clock is disabled.
-10693	extScanClockError	Cannot use an external scan clock when performing a single scan of a single channel.
-10694	unsafeSamplingFreqError	The sampling frequency exceeds the safe maximum rate for the hardware, gains, and filters used.
-10695	DMANotAllowedError	You have set up an operation that requires the use of interrupts. DMA is not allowed. For example, some DAQ events, such as messaging and LabVIEW occurrences, require interrupts.
-10696	multiRateModeError	Multi-rate scanning can not be used with AMUX-64, SCXI, or pretriggered acquisitions.
-10697	rateNotSupportedError	NI-DAQ was unable to convert your timebase/interval pair to match the actual hardware capabilities of the specified board.
-10698	timebaseConflictError	You cannot use this combination of scan and sample clock timebases for the specified board.
-10699	polarityConflictError	You cannot use this combination of scan and sample clock source polarities for this operation and board.
-10700	signalConflictError	You cannot use this combination of scan and convert clock signal sources for this operation and board.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10701	noLaterUpdateError	The call had no effect because the specified channel had not been set for later internal updates.
-10702	prePostTriggerError	Pretriggering and posttriggering cannot be used simultaneously on the Lab and 1200 series devices.
-10710	noHandshakeModeError	The specified port has not been configured for handshaking.
-10720	noEventCtrError	The specified counter is not configured for event-counting operation.
-10740	SCXITrackHoldError	A signal has already been assigned to the SCXI track-and-hold trigger line, or a control call was inappropriate because the specified module is not configured for one-channel operation.
-10780	sc2040InputModeError	When you have an SC-2040 attached to your device, all analog input channels must be configured for differential input mode.
-10781	outputTypeMustBeVoltageError	The polarity of the output channel cannot be bipolar when outputting currents.
-10782	sc2040HoldModeError	The specified operation cannot be performed with the SC-2040 configured in hold mode.
-10783	calConstPolarityConflictError	Calibration constants in the load area have a different polarity from the current configuration. Therefore, you should load constants from factory.
-10800	timeOutError	The operation could not complete within the time limit.
-10801	calibrationError	An error occurred during the calibration process.
-10802	dataNotAvailError	The requested amount of data has not yet been acquired.
-10803	transferStoppedError	The transfer has been stopped to prevent regeneration of output data.
-10804	earlyStopError	The transfer stopped prior to reaching the end of the transfer buffer.
-10805	overRunError	The clock source for the input task is faster than the maximum clock rate the device supports. If you are allowing the driver to calculate the analog input channel clock rate, the driver bases the clock rate on the board type; so you should check that your board type is correct in the configuration utility.
-10806	noTrigFoundError	No trigger value was found in the input transfer buffer.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10807	earlyTrigError	The trigger occurred before sufficient pretrigger data was acquired.
-10808	LPTCommunicationError	An error occurred in the parallel port communication with the DAQ device.
-10809	gateSignalError	Attempted to start a pulse width measurement with the pulse in the phase to be measured (for example, high phase for high-level gating).
-10840	internalDriverError	An unexpected error occurred inside the driver when performing the given operation.
-10841	firmwareError	The firmware does not support the specified operation, or the firmware operation could not complete due to a data-integrity problem.
-10842	hardwareError	The hardware is not responding to the specified operation, or the response from the hardware is not consistent with the functionality of the hardware.
-10843	underFlowError	Because of system limitations, the driver could not write data to the device fast enough to keep up with the device throughput.
-10844	underWriteError	New data was not written to the output transfer buffer before the driver attempted to transfer the data to the device.
-10845	overFlowError	Because of system limitations, the driver could not read data from the device fast enough to keep up with the device throughput; the onboard device memory reported an overflow error.
-10846	overWriteError	The driver wrote new data into the input transfer buffer before the previously acquired data was read.
-10847	dmaChainingError	New buffer information was not available at the time of the DMA chaining interrupt; DMA transfers will terminate at the end of the currently active transfer buffer.
-10848	noDMACountAvailError	The driver could not obtain a valid reading from the transfer-count register in the DMA controller.
-10849	openFileError	The configuration file could not be opened.
-10850	closeFileError	Unable to close a file.
-10851	fileSeekError	Unable to seek within a file.
-10852	readFileError	Unable to read from a file.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10853	writeFileError	Unable to write to a file.
-10854	miscFileError	An error occurred accessing a file.
-10855	osUnsupportedError	NI-DAQ does not support the current operation on this particular version of the operating system.
-10856	osError	An unexpected error occurred from the operating system while performing the given operation.
-10857	internalKernelError	An unexpected error occurred inside the kernel while performing this operation.
-10880	updateRateChangeError	A change to the update rate is not possible at this time because of one of the following reasons: <ol style="list-style-type: none"> 1. When waveform generation is in progress, you cannot change the interval timebase. 2. When you make several changes in a row, you must give each change enough time to take effect before requesting further changes.
-10881	partialTransferCompleteError	You cannot do another transfer after a successful partial transfer.
-10882	daqPollDataLossError	The data collected on the remote SCXI unit was overwritten before it could be transferred to the buffer in the host. Try using a slower data acquisition rate if possible.
-10883	wfmPollDataLossError	New data could not be transferred to the waveform buffer of the remote SCXI unit to keep up with the waveform update rate. Try using a slower waveform update rate if possible.
-10884	pretrigReorderError	Could not rearrange data after a pretrigger acquisition completed.
-10920	gpctrDataLossError	One or more data points may have been lost during buffered GPCTR operations due to the speed limitations of your system.
-10940	chassisResponseTimeoutError	No response was received from the remote SCXI unit within the specified time limit.
-10941	reprogrammingFailedError	Reprogramming the remote SCXI unit was unsuccessful. Please try again.

Table A-4. Data Acquisition VI Error Codes (Continued)

Error Code	Error Name	Description
-10942	invalidResetSignatureError	An invalid reset signature was sent from the host to the remote SCXI unit.
-10943	chassisLockupError	The interrupt service routine on the remote SCXI unit is taking longer than necessary. You do not need to reset your remote SCXI unit; however, you need to clear and restart your data acquisition.

Table A-5. AppleEvent Error Codes

Error Code	Error Name	Description
-1700	errAEO coercionFail	Data could not be coerced to the requested descriptor type.
-1701	errAEDescNotFound	Descriptor record was not found.
-1702	errAECorruptData	Data in an Apple event could not be read.
-1703	errAEWrongDataType	Wrong descriptor type.
-1704	errAENotAEDesc	Not a valid descriptor record.
-1705	errAEBadListItem	Operation involving a list item failed.
-1706	errAENewerVersion	Need a newer version of AppleEvent Manager.
-1707	errAENotAppleEvent	The event is not an Apple event.
-1708	errAERReplyNotValid	AEResetTimer was passed an invalid reply parameter.
-1709	errAERReplyNotValid	AEResetTimer was passed an invalid reply parameter.
-1710	errAEUnknownSendMode	Invalid sending mode was passed.
-1711	errAEWaitCanceled	User canceled out of wait loop for reply or receipt.
-1712	errAETimeout	Apple event timed out.
-1713	errAENoUserInteraction	No user interaction allowed.
-1714	errAENotASpecialFunction	Wrong keyword for a special function.
-1715	errAEParamMissed	Handler did not get all required parameters.
-1716	errAEUnknownAddressType	Unknown Apple event address type.
-1717	errAEHandlerNotFound	No handler in the dispatch tables fits the parameters to AEGetEventHandler or AEGetCoercionHandler.
-1718	errAERReplyNotArrived	The contents of the reply you are accessing have not arrived yet.
-1719	errAEIllegalIndex	Index is out of range in a put operation.

Table A-6. Instrument Driver Error Codes

Status	Status Description	Set By
0	No error; the call was successful	—
-1200	Invalid syntax string	VISA Transition Library
-1201	Error finding instruments	VISA Transition Library
-1202	Unable to initialize interface or instrument	VISA Transition Library
-1205	Invalid Instrument Handle	VISA Transition Library
-1210	Parameter out of range	Instrument Driver VI
-1215	Error closing instrument	VISA Transition Library
-1218	Error getting attribute	VISA Transition Library
-1219	Error setting attribute	VISA Transition Library
-1223	Instrument identification query failed	Instrument Driver Initialize VI
-1224	Error clearing instrument	VISA Transition Library
-1225	Error triggering instrument	VISA Transition Library
-1226	Error polling instrument	VISA Transition Library
-1230	Error writing to instrument	VISA Transition Library
-1231	Error reading from instrument	VISA Transition Library
-1236	Error interpreting instrument response	Instrument Driver VI
-1240	Instrument timed out	VISA Transition Library
-1250	Error mapping VXI address	VISA Transition Library
-1251	Error unmapping VXI address	VISA Transition Library
-1252	Error accessing VXI address	VISA Transition Library
-1260	Function not supported by controller	VISA Transition Library
-1300	Instrument-specific error	—
-13xx	Developer-specified error codes	—

Table A-7. PPC Error Codes

Code	Name	Description
-900	notInitErr	PPC Toolbox has not been initialized.
-902	nameTypeErr	Invalid or inappropriate locationKindSelector in locationName.
-903	noPortErr	Invalid port name. Unable to open port or bad portRefNum.
-904	noGlobalsErr	The system is unable to allocate memory. This is a critical error, and you should restart.
-905	localOnlyErr	Network activity is currently disabled.
-906	destPortErr	Port does not exist at destination.
-907	sessTableErr	PPC Toolbox is unable to create a session.
-908	noSessionErr	Invalid session reference number.
-909	badReqErr	Bad parameter or invalid state for this operation.
-910	portNameExistsErr	Another port is already open with this name (perhaps in another application).
-911	noUserNameErr	User name unknown on destination machine.
-912	userRejectErr	Destination rejected the session request.
-913	noMachineNameErr	User has not named his Macintosh in the Network Setup Control Panel.
-914	noToolboxNameErr	A system resource is missing.
-915	noResponseErr	Unable to contact destination application.
-916	portClosedErr	The port was closed.
-917	sessClosedErr	The session has closed.
-919	badPortNameErr	PPCPortRec is invalid.
-922	noDefaultUserErr	User has not specified owner name in Sharing Setup Control Panel.
-923	notLoggedInErr	The default userRefNum does not yet exist.
-924	noUserRefErr	Unable to create a new userRefNum.
-925	networkErr	An error has occurred in the network.
-926	noInformErr	PPCStart failed because destination did not have an inform pending.
-927	authFailErr	User's password is wrong.

Table A-7. PPC Error Codes (Continued)

Code	Name	Description
-928	noUserRecErr	Invalid user reference number.
-930	badServiceMesthodErr	Service method is other than ppcServiceRealTime.
-931	badLocNameErr	Location name is invalid.
-932	guestNotAllowedErr	Destination port requires authentication.

Table A-8. GPIB Error Codes

Error Code	Error Name	Description
0	EDVR	Error connecting to driver.
1	ECIC	Command requires GPIB Controller to be CIC.
2	ENOL	Write detected no Listeners.
3	EADR	GPIB Controller not addressed correctly.
4	EARG	Invalid argument or arguments.
5	ESAC	Command requires GPIB Controller to be SC.
6	EABO	I/O operation aborted.
7	ENEB	Nonexistent board.
8	EDMA	DMA hardware error detected.
9	EBTO	DMA hardware μ P bus timeout.
11	ECAP	No capability.
12	EFSO	File system operation error.
13	EOWN	Shareable board exclusively owned.
14	EBUS	GPIB bus error.
15	ESTB	Serial poll byte queue overflow.
16	ESRQ	SRQ stuck on.
17	ECMD	Unrecognized command.
19	EBNP	Board not present.
20	ETAB	Table error.
30	NADDR	No GPIB address input.
31	NSTRG	No string input (write).
32	NCNT	No count input (read).

Table A-9. LabVIEW Function Error Codes

Error Code	Error Name	Description
0	—	No error.
1	—	Manager argument error.
2	—	Argument error.
3	—	Out of zone.
4	—	End of file.
5	—	File already open.
6	—	Generic file I/O error.
7	—	File not found.
8	—	File permission error.
9	—	Disk full.
10	—	Duplicate path.
11	—	Too many files open.
12	—	System feature not enabled.
13	—	Resource file not found.
14	—	Cannot add resource.
15	—	Resource not found.
16	—	Image not found.
17	—	Image memory error.
18	—	Pen does not exist.
19	—	Config type invalid.
20	—	Config token not found.
21	—	Config parse error.
22	—	Config memory error.
23	—	Bad external code format.
24	—	Bad external code offset.
25	—	External code not present.
26	—	Null window.
27	—	Destroy window error.

Table A-9. LabVIEW Function Error Codes (Continued)

Error Code	Error Name	Description
28	—	Null menu.
29	—	Print aborted.
30	—	Bad print record.
31	—	Print driver error.
32	—	Windows error during printing.
33	—	Memory error during printing.
34	—	Print dialog error.
35	—	Generic print error.
36	—	Invalid device refnum.
37	—	Device not found.
38	—	Device parameter error.
39	—	Device unit error.
40	—	Cannot open device.
41	—	Device call aborted.
42	—	Generic error.
43	—	Cancelled by user.
44	—	Object ID too low.
45	—	Object ID too high.
46	—	Object not in heap.
47	—	Unknown heap.
48	—	Unknown object (invalid DefProc).
49	—	Unknown object (DefProc not in table).
50	—	Message out of range.
51	—	Invalid (null) method.
52	—	Unknown message.
53	—	Manager call not supported.
54	—	Bad address.
55	—	Connection in progress.
56	—	Connection timed out.

Table A-9. LabVIEW Function Error Codes (Continued)

Error Code	Error Name	Description
57	—	Connection is already in progress.
58	—	Network attribute not supported.
59	—	Network error.
60	—	Address in use.
61	—	System out of memory.
62	—	Connection aborted.
63	—	Connection refused.
64	—	Not connected.
65	—	Already connected.
66	—	Connection closed.
67	—	Initialization error (interapplication manager).
68	—	Bad occurrence.
69	—	Wait on unbound occurrence handler.
70	—	Occurrence queue overflow.
71	—	Datalog type conflict.
72	—	Unused.
73	—	Unrecognized type (interapplication manager).
74	—	Memory corrupt.
75	—	Failed to make temporary DLL.
76	—	Old CIN version.
77	—	Unknown error code.
81	—	Format specifier type mismatch.
82	—	Unknown format specifier.
83	—	Too few format specifiers.
84	—	Too many format specifiers.
85	—	Scan failed.

Table A-10. LabVIEW-Specific PPC Error Codes

Error Code	Error Name	Description
1	errNoPPCToolBox	The PPC ToolBox either does not exist (requires System 7.0 or later), or could not be initialized.
2	errNoGlobals	The CIN in the PPC VI could not get its globals.
3	errTimedOut	The PPC operation exceeded its timeout limit.
4	errAuthRequired	The target specified in the PPC Start Session VI required authentication, but the authentication dialog was not allowed.
5	errbadState	The PPC Start Session VI found itself in an unexpected state.

Table A-11. TCP and UDP Error Codes

Error Code	Error Name	Description
53	mgNotSupported	LabVIEW Manager call not supported.
54	ncBadAddressErr	The net address was ill-formed.
55	ncInProgressErr	Operation is in progress.
56	ncTimeOutErr	Operation exceeded the user-specified time limit.
57	ncBusyErr	The connection was busy.
58	ncNotSupportedErr	Function not supported.
59	ncNetErr	The network is down, unreachable, or has been reset.
60	ncAddrInUseErr	The specified address is currently in use.
61	ncSysOutOfMemErr	System could not allocate necessary memory.
62	ncSysConnAbortedErr	System caused connection to be aborted.
63	ncConnRefusedErr	Connection is not established.
65	ncAlreadyConnectedErr	Connection is already established.
66	ncConnClosedErr	Connection was closed by peer.

Table A-12. Serial Port Error Codes

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

Table A-13. LabVIEW-Specific Error Codes for AppleEvent Messages

Error Code	Error Name	Description
1000	kLVE_InvalidState	The VI is in a state that does not allow it to run.
1001	kLVE_FPNotOpen	The VI front panel is not open.
1002	kLVE_CtrlErr	The VI has controls on its front panel that are in an error state.
1003	kLVE_VIBad	The VI is broken.
1004	kLVE_NotInMem	The VI is not in memory.

Table A-14. DDE Error Codes

Error Code	Error Name	Description
00000	—	No error.
14001	DDE_INVALID_REFNUM	Invalid refnum.
14002	DDE_INVALID_STRING	Invalid string.
14003	DDEML_ADVACKTIMEOUT	Request for a synchronous advise transaction has timed out.
14004	DDEML_BUSY	Response set the DDE_FBUSY bit.
14005	DDEML_DATAACKTIMEOUT	Request for a synchronous data transaction has timed out.
14006	DDEML_DDL_NOT_INITIALIZED	DDEML called without first calling <code>DdeInitialize</code> , or was passed an invalid instance identifier.
14007	DDEML_DLL_USAGE	A monitor or client-only application has attempted a DDE transaction.

Table A-14. DDE Error Codes (Continued)

Error Code	Error Name	Description
14008	DDEML_EXECACKTIMEOUT	Request for a synchronous execute transaction has timed out.
14009	DDEML_INVALIDPARAMETER	Parameter not validated by the DDML.
14010	DDEML_LOW_MEMORY	Server application has outrun client, consuming large amounts of memory.
14011	DDEML_MEMORY_ERROR	A memory allocation failed.
14012	DDEML_NOTPROCESSED	Request or poke is for an invalid istem.
14013	DDEML_NO_CONV_ESTABLISHED	Client conversation attempt failed.
14014	DDEML_POKEACTIMEOUT	Transaction failed.
14015	DDEML_POSTMSG_FAILED	Request for a synchronous poke transaction has timed out.
14016	DDEML_REENTRANCY	An application with a synchronous transaction in progress attempted to initiate another transaction, or a DDEML callback function called <code>DdeEnableCallback</code> .
14017	DDEML_SERVER_DIED	Server-side transaction attempted on conversation terminated by client, or service terminated before completing a transaction.
14018	DDEML_SYS_ERROR	Internal error in the DDMEML.
14019	DDEML_UNADVACKTIMEOUT	Request to end advise has timed out.
14020	DDEML_UNFOUND_QUEUE_ID	Invalid transaction identifier passed to DDEML function.
14021	—	Invalid command code.
14022	—	Occurrence timeout; the transaction timed out before it completed.

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 B-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 AC, DC (for PCI-6110E, PCI-6111E)
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 B-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
PCI-6110E PCI-6111E	4 DI 2 DI	12 bits 12 bits	0.2, 0.5, 1.2, 5, 10, 20, 50	±10	512	Up to 4 Up to 2
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
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 bits	1, 2, 5, 10, 20, 50, 100	±10, 0 to 10	512	Up to 512

Table B-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
AT-MIO-16(L) AT-MIO-16(H) AT-MIO-16D(L) AT-MIO-16D(H)	16SE, 8DI	12 bits	(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 Chapter 3, *Basic LabVIEW Data Acquisition Concepts*, 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 B-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 PCI-6110E PCI-6111E	SW, Pre, Post, (and Analog on E-1, E-2, E-3, E-4, PCI-6110E, and PCI-6111E)	E-1: 1 M, E-2 and E-3: 500 k, E-4: 250 k, E-10 and DE-10: 100 k PCI-6110E and PCI-6111E: 5M	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

Table B-3. Analog Input Characteristics—MIO and AI Devices (Part 2) (Continued)

Device	Triggers ¹	Max Sampling Rate (S/s)	Transfer Method
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 actually is done in the interrupt service routine (interrupts only) and is different than conditional retrieval.

Table B-4. Internal Channel Support—MIO and AI Devices

Device	Internal Channels
AT-MIO-16XE10, AT-MIO-16XE-50, NEC-MIO-16XE-50, DAQPad-MIO-16XE-50	IntAIGnd, IntRef5V, IntAOGndVsAIGnd, IntAOCh0, IntAOCh0VsRef5V, IntA0Ch1, IntAOCh1VsRef5V
DAQCard-AI-16E-4, NEC-AI-16E-4	IntAIGnd, IntRef5V, IntA0GndVsAIGnd
PCI-MIO-16XE-10, PCI-MIO-16XE-50, PXI-6030E, PXI-6011E, PCI-6031, CPCI-6030E, CPCI-6011E, VXI-MIO-64XE-10	IntAIGnd, IntRef5V, IntAOGndVsAIGnd, IntAOch0, IntAOCh0VsRef5V, IntAOCh1, IntAOCh1VsRef5V, IntAOChVsAOch0, IntDevTemp
PCI-MIO-16E-1, PCI-MIO-16E-4, PXI-6070E, PXI-6040E, PCI-6071E, CPCI-6070E, CPCI-6040E, VXI-MIO-64E-1	IntAIGnd, IntRef5V, IntCmRef5V, IntAOGndVsAIGnd, IntAOCh0, IntAOCh0VsRef5V, IntAOCh1, IntAOCh1VsRef5V, IntAOCh1VsAOCh0, IntDevTemp
AT-AI-16XE-10, PCI-6032E, PCI-6033E, DAQCard-AI-16XE-50, NEC-AI-16XE-50	IntAIGnd, IntRef5V, IntA0GndVsAIGnd
AT-MIO-16E-1, AT-MIO-16E-2, AT-MIO-16E-3, AT-MIO-16DE-10, AT-MIO-16E-10, DAQPad-6020E, NEC-MIO-16E-4	IntAIGnd, IntRef5V, IntCmRef5V, IntAOGndVsAIGnd, IntAOCh0, IntAOCh0VsRef5V, IntAOCh1, IntAOCh1VsRef5V

Table B-5. Analog Output Characteristics—MIO and AI Devices

Device	Channel Numbers	DAC Type	FIFO Size	Output Limits (V)	Update Clocks	Transfer Method
AT-MIO-16E-1 AT-MIO-16E-2 AT-MIO-64E-3 NEC-MIO-15E-4 VXI-MIO-64E-1	0, 1	12-bit double buffered	2048	0 to 10, ± 10 , $\pm V_{ref}$, 0 to V_{ref}	Update clock 1 or external update.	DMA, interrupts
AT-MIO-16E-10	—	12-bit double buffered	0	± 10 , 0 to 10	Update clock 1 or external update.	DMA, interrupts
AT-MIO-XE-50 NEC-MIO-16XE-50	—	12-bit double buffered	0	± 10	Update clock 1 or external update.	DMA, interrupts
DAQPad-MIO-16XE-50	—	12-bit double buffered	0	± 10	Update clock 1 or external update.	Interrupts
AT-MIO-16XE-10 VXI-MIO-64XE-10	—	16-bit double buffered	0	± 10 , 0 to 10	Update clock 1 or external update.	DMA, interrupts
OCI-MIO-16E-1 CPCI-6070E PXI-6070E PCI-6071E	—	12-bit double buffered	2048	± 10 , 0 to 10	Update clock 1 or external update.	DMA, interrupts
PCI-MIO-16E-4 CPCI-6040E PXI-6040E	—	12-bit double buffered	512	± 10 , 0 to 10	Update clock 1 or external update.	DMA, interrupts
PCI-MIO-16XE-50 CPCI-6011E PXI-6011E	—	12-bit double buffered	0	± 10	Update clock 1 or external update.	DMA, interrupts

Table B-5. Analog Output Characteristics—MIO and AI Devices (Continued)

Device	Channel Numbers	DAC Type	FIFO Size	Output Limits (V)	Update Clocks	Transfer Method
PCI-MIO-16XE-10 CPCI-6030E PXI-6030E PCI-6031E	—	16-bit double buffered	5048	± 10 , 0 to 10	Update clock 1 or external update.	DMA, interrupts
DAQPad-6020E	—	12-bit double buffered	0	± 10 , 0 to 10	Update clock 1 or external update.	DMA, interrupts
PCI-6110E PCI-6111E	—	16-bit double buffered	4096	± 5	Update clock 1 or external update.	DMA, interrupts
AT-MIO-16F-5 AT-MIO-64F-5	0, 1	12-bit double buffered (64F-5: 2K FIFO)	—	0 to 10, ± 10 , $\pm V_{ref}$, 0 to V_{ref}	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.	DMA, interrupts
AI-MIO-16X	0, 1	16-bit double buffered (2K FIFO)	—	± 10 , 0 to 10, $\pm V_{ref}$, 0 to V_{ref}	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.	DMA, interrupts

Table B-5. Analog Output Characteristics—MIO and AI Devices (Continued)

Device	Channel Numbers	DAC Type	FIFO Size	Output Limits (V)	Update Clocks	Transfer Method
AT-MIO-16/16D	0, 1	12-bit double buffered	—	0 to 10, ± 10 , $\pm V_{ref}$, 0 to V_{ref}	Update clock 1 is ctr2 or external update. Timebase signal range is 1,000,000, 100,000, 10,000, 1,000, and 100.	Interrupts
NB-MIO-16/16X	0, 1	MIO-16:12-bit ; MIO-16X: 12-bit double buffered	—	0 to 10, ± 10 , $\pm V_{ref}$, 0 to V_{ref}	Update clock 1, external update (MIO-16X only). Timebase signal range is 1,000,000, 100,000, 10,000, 1,000, and 100.	MIO-16:DMA; MIO-16X: DMA, interrupts

Table B-6. Analog Output Characteristics—E-Series Devices

Device	Reglitching Capable	Ground Reference Capable	Can Control FIFO Request Modes	AO Gating, Pause/Resume Supported	Multiple Buffers Supported
AT-MIO-16E-1 AT-MIO-16E-2 AT-MIO-64E-3 NEC-MIO-16E VXI-MIO-64E-1	Yes	No	No	No	Yes
AT-MIO-16E-10 AT-MIO-16DE-10	No	No	No	Yes	Yes
AT-MIO-XE-50 NEC-MIO-16XE-50	No	No	No	No	Yes
DAQPad-MIO-16XE-50	No	No	No	No	Yes

Table B-6. Analog Output Characteristics—E-Series Devices (Continued)

Device	Reglitching Capable	Ground Reference Capable	Can Control FIFO Request Modes	AO Gating, Pause/Resume Supported	Multiple Buffers Supported
AT-MIO-16XE-10	No	No	No	Yes	Yes
VXI-MIO-64XE-10	No	No	No	No	Yes
PCI-MIO-16E-1 CPCI-6070E PXI-6070E PCI-6071E	Yes	Yes	Yes	Yes	No
PCI-MIO-16E-4 CPCI-6040E PXI-6011E	No	Yes	Yes	Yes	No
PCI-MIO-16XE-10 CPCI-6030E PXI-6030E PCI-6031E	No	No	Yes	Yes	No
DAQPad-6020E	No	Yes	No	Yes	No
PCI-6110E PCI-6111E	No	No	Yes	Yes	No

Table B-7. 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-12 AT-MIO-16DE-10 ¹ AT-MIO-16XE-50 DAQPad-MIO-16XE-50 PCI-MIO-16XE-50 PXI-6040E (MIO-16E-4) PXI-6070E (MIO-16E-1) PXI-6071E (MIO-64E-1) PCI-6031E (MIO-64XE-10) PCI-6032E (AI-16XE-10) PCI-6033E (AI-64XE-10) PCI-6110E/PCI-6111E	8-bit ports	0	No handshaking	Bit-wise direction control	None	Software polling

Table B-7. Digital I/O Hardware Capabilities—MIO and AI Devices (Continued)

Device	Port Type	Port Numbers	Handshake Modes	Direction	DIO Clocks	Transfer Method
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 bidirectional	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 B-8. Counter Characteristics—MIO and AI Devices

Device	Counter Chip Used	# of General Purpose Counters Available	Timebases Available	Number of Bits	Gate Modes Available	Outputs 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

¹SW = Software; HW = Hardware.

Table B-9. 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 B-10. 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-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 B-11. 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 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 Chapter 3, *Basic LabVIEW Data Acquisition Concepts*, in the *LabVIEW Data Acquisition Basics Manual*.

Table B-12. 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 $N \leq 7$	Software trigger, pretrigger, and posttrigger with digital trigger	62.5 k	Interrupts
Lab-PC+ SCXI-1200 DAQPad-1200 DAQCard-1200 PCI-1200	Any single channel; for multiple channels, N through 0, where $N \leq 7$.	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 $N \leq 7$	Software trigger only	50 k	Interrupts

Table B-12. 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 B-13. Analog Output Characteristics—Lab and 1200 Series and Portable Devices

Device	Channel Numbers	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 B-14. 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

Table B-14. Counter Usage for Analog Input and Output—Lab Series and Portable Devices (Continued)

Device Name	Counter Chip Used	AI Channel Clock	AI Sample Counter	AI Scan Clock	AO Update Clock
DAQCard 516 PC-516	82C54	Ctr 0	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 B-15. 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 DAQCard-516	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 B-16. 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 - DDS Mode	256 samples minimum Memory depth maximum Note: Buffer size should be a multiple of 8 samples. 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
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%.

Table B-16. Analog Output and Digital Output Characteristics—54XX Series Devices (Continued)

Characteristics	AT-5411, PCI-5411
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 B-17. Counter/Timer Characteristics—Lab and 1200 Series and Portable Devices

Device	Counter Chip Used	Number 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 28, 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 28, Advanced Counter VIs .	Down

SCXI Module Hardware Capabilities

Table B-18. 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
SCXI-1120 SCXI-1121	8 DI (SCXI-1120) 4 DI (SCXI-1121)	±5	1, 2, 5, 10, 20, 50, 100, 200, 500, 1,000, 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

Table B-18. 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-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

¹DS/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

²The SCXI-1141 has an automatic filter setting. LabVIEW sets the filter frequency based on the scan rates used with the module.

Table B-19. Analog Output Characteristics—SCXI Modules

Module	Number of Channels	Output Voltage Range (V or mA)	Mode Support
SCXI-1124	6 voltage or current	0 to 1, 0 to 5, 0 to 10, ±1, ±5, ±10 (software-selectable) or 0 to 20 mA	Multiplexed

Table B-20. 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 B-21. 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.
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.

¹Functionally equivalent to the SCXI-1163, but incorporates solid-state relays in place of digital outputs.

Table B-22. Terminal Block Selection Guide—SCXI Modules

SCXI Module	Terminal Blocks	Cold-Junction Compensation Sensor (CJC)
SCXI-1100	SCXI-1303	Thermistor
SCXI-1102	SCXI-1300	IC Sensor
SCXI-1120	SCXI-1320	IC Sensor
SCXI-1121	SCXI-1321 ¹	IC Sensor
	SCXI-1327	Thermistor
	SCXI-1328	Thermistor
SCXI-1122	SCXI-1322	Thermistor
SCXI-1124	SCXI-1325	—
SCXI-1140	SCXI-1301	—
	SCXI-1304	—
SCXI-1141	SCXI-1304	—

Table B-22. Terminal Block Selection Guide—SCXI Modules (Continued)

SCXI Module	Terminal Blocks	Cold-Junction Compensation Sensor (CJC)
SCXI-1160	SCXI-1324	—
SCXI-1161	None—screw terminals located in module.	—
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 B-23. Analog Input Configuration Programmability

Device	Gain	Coupling
5102 devices	By Channel	By Channel

Table B-24. 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 B-25. Analog Output Characteristics—Analog Output Only Devices

Device	Channel Numbers	DAC Type	Output Limits	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	$\pm 10V$, $\pm V_{ref1}$, 0 to 10 V, 0 to V_{ref1} , 4 to 20 mA	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: $N \neq 6$, $N \neq 10$ *	Update clock 1 channels: DMA, interrupts; update clock 2 channels: interrupts
PC-AO-2DC (Plug and Play)	0, 1	—	0 to 10V, $\pm 5V$, 0–20mA sink software-selectable	—	—	—
DAQCard-AO-2DC	0, 1	—	0 to 10V, $\pm 5V$, 0–20mA sink software-selectable	—	—	—

*AT-AO-10 only

Dynamic Signal Acquisition Devices

Hardware Capabilities

Table B-26. 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 B-27. Analog Output Characteristics—Dynamic Signal Acquisition Devices

Device	Channel Numbers	DAC Type	Output Limits	Update Clocks	Transfer Method
PCI-4451 PCI-4552	0, 1	18-bit	±10V, ±1V, ±100	Update clock 1	DMA

Table B-28. 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:51; 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 2150:51.2k	DMA, interrupts

Digital Only Devices Hardware Capabilities

Table B-29. Digital Hardware Capabilities—Digital I/O Devices

Device	Port Type	Port Numbers	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

Table B-29. Digital Hardware Capabilities—Digital I/O Devices (Continued)

Device	Port Type	Port Numbers	Handshake Modes	Direction	DIO Clocks	Transfer Method
	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 DAQPad-6507	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 B-30. 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 B-31. Counter/Timer Characteristics—Timing Only Devices

Device	Counter Chip Used	# of General Purpose Counters Available	Timebases Available	Number of Bits	Gate Modes Available	Outputs Available	Output Modes Available	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 B-32. Analog Input Configuration Programmability

Device	Gain	Coupling
5102 devices	By Channel	By Channels

Table B-33. 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 B-34. Analog Input Characteristics, Part 2

Device	Triggers	Maximum Sampling Rate (S/s)
5102 devices	SW, Pre, Post, Analog	20,000,000 real time



GPIB Multiline Interface Messages

This appendix lists multiline interface messages, which are commands that IEEE 488 defines. Multiline interface messages manage the GPIB—they perform tasks such as initializing the bus, addressing and unaddressing devices, and setting device modes for local or remote programming. These multiline interface messages are sent and received with ATN TRUE. 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*.

Multiline Interface Messages

Hex	Oct	Dec	ASCII	Msg
00	000	0	NUL	—
01	001	1	SOH	GTL
02	002	2	STX	—
03	003	3	ETX	—
04	004	4	EOT	SDC
05	005	5	ENQ	PPC
06	006	6	ACK	—
07	007	7	BEL	—
08	010	8	BS	GET
09	011	9	HT	TCT
0A	012	10	LF	—

Hex	Oct	Dec	ASCII	Msg
0B	013	11	VT	—
0C	014	12	FF	—
0D	015	13	CR	—
0E	016	14	SO	—
0F	017	15	SI	—
10	020	16	DLE	—
11	021	17	DC1	LLO
12	022	18	DC2	—
13	023	19	DC3	—
14	024	20	DC4	DCL
15	025	21	NAK	PPU
16	026	22	SYN	—
17	027	23	ETB	—
18	030	24	CAN	SPE
19	031	25	EM	SPD
1A	032	26	SUB	—
1B	033	27	ESC	—
1C	034	28	FS	—
1D	035	29	GS	—
1E	036	30	RS	—
1F	037	31	US	—
20	040	32	SP	MLA0
21	041	33	!	MLA1
22	042	34	“	MLA2
23	043	35	#	MLA3
24	044	36	\$	MLA4

Hex	Oct	Dec	ASCII	Msg
25	045	37	%	MLA5
26	046	38	&	MLA6
27	047	39	'	MLA7
28	050	40	(MLA8
29	051	41)	MLA9
2A	052	42	*	MLA10
2B	053	43	+	MLA11
2C	054	44	,	MLA12
2D	055	45	-	MLA13
2E	056	46	.	MLA14
2F	057	47	/	MLA15
30	060	48	0	MLA16
31	061	49	1	MLA17
32	062	50	2	MLA18
33	063	51	3	MLA19
34	064	52	4	MLA20
35	065	53	5	MLA21
36	066	54	6	MLA22
37	067	55	7	MLA23
38	070	56	8	MLA24
39	071	57	9	MLA25
3A	072	58	:	MLA26
3B	073	59	;	MLA27
3C	074	60	<	MLA28
3D	075	61	=	MLA29
3E	076	62	>	MLA30

Hex	Oct	Dec	ASCII	Msg
3F	077	63	?	UNL
40	100	64	@	MTA0
41	101	65	A	MTA1
42	102	66	B	MTA2
43	103	67	C	MTA3
44	104	68	D	MTA4
45	105	69	E	MTA5
46	106	70	F	MTA6
47	107	71	G	MTA7
48	110	72	H	MTA8
49	111	73	I	MTA9
4A	112	74	J	MTA10
4B	113	75	K	MTA11
4C	114	76	L	MTA12
4D	115	77	M	MTA13
4E	116	78	N	MTA14
4F	117	79	O	MTA15
50	120	80	P	MTA16
51	121	81	Q	MTA17
52	122	82	R	MTA18
53	123	83	S	MTA19
54	124	84	T	MTA20
55	125	85	U	MTA21
56	126	86	V	MTA22
57	127	87	W	MTA23
58	130	88	X	MTA24

Hex	Oct	Dec	ASCII	Msg
59	131	89	Y	MTA25
5A	132	90	Z	MTA26
5B	133	91	[MTA27
5C	134	92	\	MTA28
5D	135	93]	MTA29
5E	136	94	^	MTA30
5F	137	95	_	UNT
60	140	96	`	MSA0,PPE
61	141	97	a	MSA1,PPE
62	142	98	b	MSA2,PPE
63	143	99	c	MSA3,PPE
64	144	100	d	MSA4,PPE
65	145	101	e	MSA5,PPE
66	146	102	f	MSA6,PPE
67	147	103	g	MSA7,PPE
68	150	104	h	MSA8,PPE
69	151	105	i	MSA9,PPE
6A	152	106	j	MSA10,PPE
6B	153	107	k	MSA11,PPE
6C	154	108	l	MSA12,PPE
6D	155	109	m	MSA13,PPE
6E	156	110	n	MSA14,PPE
6F	157	111	o	MSA15,PPE
70	160	112	p	MSA16,PPD
71	161	113	q	MSA17,PPD
72	162	114	r	MSA18,PPD

Hex	Oct	Dec	ASCII	Msg
73	163	115	s	MSA19,PPD
74	164	116	t	MSA20,PPD
75	165	117	u	MSA21,PPD
76	166	118	v	MSA22,PPD
77	167	119	w	MSA23,PPD
78	170	120	x	MSA24,PPD
79	171	121	y	MSA25,PPD
7A	172	122	z	MSA26,PPD
7B	173	123	{	MSA27,PPD
7C	174	124		MSA28,PPD
7D	175	125	}	MSA29,PPD
7E	176	126	~	MSA30,PPD
7F	177	127	DEL	—

Message Definitions

Mnemonics	Definition
DCL	Device Clear
GET	Group Execute Trigger
GTL	Go To Local
LLO	Local Lockout
MLA	My Listen Address
MSA	My Secondary Address
MTA	My Talk Address
PPC	Parallel Poll Configure
PPD	Parallel Poll Disable
PPE	Parallel Poll Enable

Mnemonics	Definition
PPU	Parallel Poll Unconfigure
SDC	Selected Device Clear
SPD	Serial Poll Disable
SPE	Serial Poll Enable
TCT	Take Control
UNL	Unlisten
UNT	Untalk

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 e-mail 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 USA 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.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
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	09 725 725 11	09 725 725 55
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 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
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
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

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 Manual*

Edition Date: January 1998

Part Number: 321526B-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 _____

E-Mail Address _____

Phone (____) _____ Fax (____) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 78730-5039

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Index

Numbers/Symbols

1200 Calibrate, 29-2
1D ANOVA, 44-2
1D Linear Evaluation, 46-2
1D Polar To Rectangular, 46-2
1D Polynomial Evaluation, 46-2
1D Rectangular To Polar, 46-3
2D ANOVA, 44-3
2D Linear Evaluation, 46-3
2D Polynomial Evaluation, 46-4
3D ANOVA, 44-4

A

A x B, 45-2
A x Vector, 45-2
A2000 Calibrate, 29-3
A2000 Configure, 29-4
A2100 Calibrate, 29-4
A2100 Config, 29-5
A2150 Calibrate (Macintosh), 29-6
A2150 Config, 29-5
AC & DC Estimator, 40-2
Access Rights, 11-14
Acquire Semaphore, 13-17
ActiveX Variant to G Data, 51-4
Additional User Definable Constants, 4-21
Adjacent Counters, 27-2
AECreat Comp Descriptor, 52-14
AECreat Descriptor List, 52-15
AECreat Logical Descriptor, 52-14
AECreat Object Specifier, 52-14
AECreat Range Descriptor, 52-15
AECreat Record, 52-15
AESend, 52-13
AESend Abort VI, 52-8
AESend Close VI, 52-8
AESend Do Script, 52-6
AESend Finder Open, 52-6
AESend Open, 52-7
AESend Open Document, 52-7
AESend Open, Run, Close VI, 52-8
AESend Print Document, 52-7
AESend Quit Application, 52-7
AESend Run VI, 52-9
AESend VI Active?, 52-9
AI Acquire Waveform, 15-1
AI Acquire Waveforms, 15-2
AI Buffer Config, 18-1
AI Buffer Read, 18-2
AI Clear, 16-2
AI Clock Config, 18-3
AI Config, 16-2
AI Continuous Scan, 17-2
AI Control, 18-5
AI Group Config, 18-6
AI Hardware Config, 18-8
AI Parameter, 18-12
AI Read, 16-3
AI Read One Scan, 17-3
AI Sample Channel, 15-2
AI Sample Channels, 15-3
AI Single Scan (Intermediate), 16-3
AI SingleScan (Advanced), 18-13
AI Start, 16-4
AI Trigger Config, 18-14
AI Waveform Scan, 17-4
AllSPoll, 35-4
Amplitude and Phase Spectrum, 40-2
And, 5-2
And Array Elements, 5-3
AO Buffer Config, 22-1
AO Buffer Write, 22-2
AO Clear, 20-2
AO Clock Config, 22-3
AO Config, 20-2
AO Continuous Gen, 21-2
AO Control, 22-3
AO Generate Waveform, 19-1
AO Generate Waveforms, 19-2
AO Group Config, 22-3
AO Hardware Config, 22-4
AO Parameter, 22-4
AO Single Update, 22-4
AO Start, 20-3
AO Trigger and Gate Config (Windows), 22-4
AO Update Channel, 19-2
AO Update Channels, 19-2

AO Wait, 20-3
AO Waveform Gen, 21-4
AO Write, 20-3
AO Write One Update, 21-5
AO-6/10 Calibrate (Windows), 29-6
Arbitrary Wave, 38-2
Array Max & Min, 7-3
Array Of Strings To Path, 6-19, 11-15
Array Size, 7-4
Array Subset, 7-4
Array To Cluster, 7-4, 8-4
Array To Spreadsheet String, 6-6
Auto Power Spectrum, 40-2
AutoCorrelation, 39-2

B

Beep, 13-2
Bessel Coefficients, 41-2
Bessel Filter, 41-2
Blackman Window, 42-2
Blackman-Harris Window, 42-2
Boolean Array To Number, 4-10, 5-3
Boolean Constant, 5-5
Boolean To (0,1), 4-10, 5-3
Build Array, 7-4
Build Cluster Array, 8-4
Build Path, 11-6
Bundle, 8-4
Bundle By Name, 8-5
Butterworth Coefficients, 41-3
Butterworth Filter, 41-3
Byte Array To String, 4-10, 6-19
Bytes at Serial Port, 36-1

C

cac – Become active Controller, 34-9
Call By Reference Node, 12-2
Call Chain, 12-3
Call Library Function, 13-3
Cancel Notification, 13-9
Carriage Return, 6-20
Cascade→Direct Coefficients, 41-3
Case Structure, 3-2
Cast Unit Bases, 4-11
Channel To Index, 29-7
Chebyshev Coefficients, 41-4

Chebyshev Filter, 41-4
Chi Square Distribution, 44-5
Chirp Pattern, 38-3
Cholesky Factorization, 45-3
Close All PPC Ports, 53-2
Close Application or VI Reference, 12-3
Close Automation Refnum, 51-2
Close Config Data, 11-22
Close File, 11-6
Cluster To Array, 7-5, 8-5
cmd – Send IEEE488 commands, 34-9
Code Interface Node, 13-2
Complex A x B, 45-3
Complex A x Vector, 45-4
Complex Cholesky Factorization, 45-4
Complex Conjugate, 4-20
Complex Determinant, 45-4
Complex Dot Product, 45-5
Complex Eigenvalues & Vectors, 45-6
Complex FFT, 39-3
Complex Inverse Matrix, 45-6
Complex LU Factorization, 45-7
Complex Matrix Condition Number, 45-7
Complex Matrix Norm, 45-7
Complex Matrix Rank, 45-8
Complex Matrix Trace, 45-8
Complex Outer Product, 45-8
Complex Polynomial Roots, 47-1
Complex PseudoInverse Matrix, 45-9
Complex QR Factorization, 45-9
Complex SVD Factorization, 45-10
Complex To Polar, 4-20
Complex To Re/Im, 4-20
Compound Arithmetic, 5-3
Concatenate Strings, 6-6
Contingency Table, 44-5
Continuous Pulse Generator Config, 27-3
Control Help Window, 12-7
Control Online Help, 12-7
Convert RTD Reading, 30-2
Convert Strain Gauge Reading, 30-3
Convert Thermistor Reading, 30-6
Convert Thermocouple Buffer, 30-8
Convert Thermocouple Reading, 30-8
Convert Unit, 4-11
Convolution, 39-4, 41-4
Copy, 11-15
Cosecant, 4-14

Cosine, 4-14
 Cosine Tapered Window, 42-3
 Cotangent, 4-14
 Count Events or Time, 26-2
 Counter Read, 27-3
 Counter Start, 27-3
 Counter Stop, 27-3
 Create Notifier, 13-9
 Create Queue, 13-12
 Create Rendezvous, 13-15
 Create Semaphore, 13-17
 Create Special Complex Matrix, 45-10
 Create Special Matrix, 45-10
 Creating AppleEvent Parameters Using
 Object Specifiers, 52-14
 Cross Power, 39-5
 Cross Power Spectrum, 40-3
 CrossCorrelation, 39-6
 CTR Buffer Config, 28-2
 CTR Buffer Read, 28-2
 CTR Control, 28-10
 CTR Group Config, 28-3
 CTR Mode Config, 28-3
 CTR Pulse Config, 28-9
 Current VI's Path Constant, 11-26

D

DAQ Occurrence Config (Windows), 29-9
 Date/Time To Seconds, 10-6
 DDE Advise Check, 50-2
 DDE Advise Start, 50-2
 DDE Advise Stop, 50-2
 DDE Close Conversation, 50-2
 DDE Execute, 50-2
 DDE Open Conversation, 50-3
 DDE Poke, 50-3
 DDE Request, 50-3
 DDE Srv Check Item, 50-3
 DDE Srv Register Item, 50-4
 DDE Srv Register Service, 50-4
 DDE Srv Set Item, 50-4
 DDE Srv Unregister Item, 50-4
 DDE Srv Unregister Service, 50-4
 Decimal Digit?, 9-6
 Decimate, 39-8
 Decimate 1D Array, 7-5
 Deconvolution, 39-9

Default Directory Constant, 11-27
 Delayed Pulse Generator Config, 27-4
 Delete, 11-15
 Delete Menu Items, 12-8
 Derivative $x(t)$, 39-10
 Destroy Notifier, 13-10
 Destroy Queue, 13-13
 Destroy Rendezvous, 13-15
 Destroy Semaphore, 13-18
 Determinant, 45-10
 DevClear, 35-2
 DevClearList, 35-4
 Device Reset, 29-10
 Digital Buffer Config, 25-3
 Digital Buffer Control, 25-3
 Digital Buffer Read, 25-3
 Digital Buffer Write, 25-3
 Digital Clock Config, 25-4
 Digital Group Config, 25-4
 Digital Mode Config, 25-5
 Digital Single Read, 25-6
 Digital Single Write, 25-6
 Digital Trigger Config, 25-7
 DIO Clear, 24-2
 DIO Config, 24-3
 DIO Parameter, 25-5
 DIO Port Config, 25-2
 DIO Port Read, 25-2
 DIO Port Write, 25-2
 DIO Read, 24-3
 DIO Single Read/Write, 24-3
 DIO Start, 24-4
 DIO Wait, 24-4
 DIO Write, 24-4
 dma – Set DMA mode or programmed I/O mode, 34-10
 Dot Product, 45-11
 Down Counter or Divider Config, 27-4
 DSA Calibrate, 29-10
 DSP2200 Calibrate (Windows), 29-11
 DSP2200 Configure (Windows), 29-11

E

Easy VISA Find Resources, 33-4
 Easy VISA Read, 33-4
 Easy VISA Serial Write and Read, 33-4
 Easy VISA Write, 33-5
 Easy VISA Write and Read, 33-5

EigenValues & Vectors, 45-11
Elliptic Coefficients, 41-5
Elliptic Filter, 41-5
Empty Path, 11-27
Empty String, 6-20
Empty String/Path?, 9-6
Enable Menu Tracking, 12-9
EnableLocal, 35-4
EnableRemote, 35-5
End of Line, 6-20
EOF, 11-15
Equal To 0?, 9-6
Equal?, 9-6
Equiripple BandPass, 41-5
Equiripple BandStop, 41-6
Equiripple HighPass, 41-6
Equiripple LowPass, 41-7
erf(x), 44-6
erfc(x), 44-6
E-Series Calibrate, 29-12
Event or Time Counter Config, 27-5
Exact Blackman Window, 42-3
Exclusive Or, 5-4
Exponential, 4-18
Exponential (Arg) -1, 4-18
Exponential Fit, 43-2
Exponential Fit Coefficients, 43-2
Exponential Window, 42-4

F

F Distribution, 44-6
Fast Hilbert Transform, 39-10
FHT, 39-12
File Dialog, 11-16
File/Directory Info, 11-16
Find First Error, 10-10
FindLstn, 35-6
FindRQS, 35-5
FIR Narrowband Coefficients, 41-7
FIR Narrowband Filter, 41-9
FIR Windowed Coefficients, 41-9
FIR Windowed Filter, 41-9
Fixed Constants, 4-22
Flat Top Window, 42-4
Flatten To String, 13-4
Flush File, 11-16
Flush Queue, 13-13

Flush Serial Buffer, 33-18
For Loop, 3-2
Force Window, 42-4
Format & Append, 6-15
Format & Strip, 6-16
Format Date/Time String Function, 10-6
Format Into String, 6-6
Formula Node, 3-3
From Decimal, 6-16
From Exponential/Fract/Eng, 6-16
From Hexadecimal, 6-16
From Octal, 6-17

G

Gaussian White Noise, 38-3
General Cosine Window, 42-5
General Error Handler, 10-11
General Histogram, 44-6
General LS Linear Fit, 43-3
General Polynomial Fit, 43-3
Generate Delayed Pulse, 26-2
Generate Occurrence, 13-19
Generate Pulse Train, 26-3
Get Channel Information, 29-19
Get DAQ Channel Names, 29-18
Get DAQ Device Information, 29-13
Get Date/Time In Seconds, 10-8
Get Date/Time String, 10-8
Get Help Window Status, 12-7
Get Menu Item Info, 12-9
Get Menu Selection, 12-9
Get Menu Shortcut Info, 12-10
Get Notifier Status, 13-10
Get Queue Status, 13-13
Get Rendezvous Status, 13-15
Get Scale Information, 29-19
Get SCXI Information, 29-14
Get Semaphore Status, 13-18
Get Target ID, 52-4, 53-3
Global Variable, 3-3
GPIB Clear, 34-3
GPIB Initialization, 34-4
GPIB Misc, 34-4
GPIB Read, 34-5
GPIB Serial Poll, 34-5
GPIB Status, 34-6
GPIB Trigger, 34-6

GPIB Wait, 34-6
 GPIB Write, 34-6
 Greater Or Equal To 0?, 9-7
 Greater Or Equal?, 9-7
 Greater Than 0?, 9-7
 Greater?, 9-6
 gts – Go from active Controller to standby, 34-10

H

Hamming Window, 42-6
 Hanning Window, 42-6
 Harmonic Analyzer, 40-3
 Hex Digit?, 9-7
 Histogram, 44-7
 Hyperbolic Cosine, 4-14
 Hyperbolic Sine, 4-15
 Hyperbolic Tangent, 4-15

I

ICTR Control, 27-5, 28-10
 IIR Cascade Filter, 41-10
 IIR Cascade Filter with Integrated Circuit, 41-11
 IIR Filter, 41-11
 IIR Filter with Integrated Circuit, 41-11
 Implies, 5-4
 Impulse Pattern, 38-4
 Impulse Response Function, 40-4
 In Port (Windows 3.1 and Windows 95), 13-7
 In Range?, 9-7
 Index & Append, 6-8
 Index & Bundle Cluster Array, 8-5
 Index & Strip, 6-8
 Index Array, 7-5
 Initialize Array, 7-5
 Insert Menu Items, 12-10
 Insert Queue Element, 13-13
 Integral $x(t)$, 39-13
 Interleave 1D Arrays, 7-6
 Interpolate 1D Array, 7-6
 Inv Chebyshev Coefficients, 41-12
 Inv Chi Square Distribution, 44-8
 Inv F Distribution, 44-9
 Inv Normal Distribution, 44-9
 Inv T Distribution, 44-9
 Inverse Chebyshev Filter, 41-12
 Inverse Complex FFT, 39-13

Inverse Cosine, 4-15
 Inverse Fast Hilbert Transform, 39-14
 Inverse FHT, 39-15
 Inverse Hyperbolic Cosine, 4-15
 Inverse Hyperbolic Sine, 4-15
 Inverse Hyperbolic Tangent, 4-16
 Inverse Matrix, 45-12
 Inverse Real FFT, 39-16
 Inverse Sine, 4-16
 Inverse Tangent, 4-16
 Inverse Tangent (2 Input), 4-16
 Invoke Node, 12-3, 51-3
 IP To String, 48-3
 ist – Set individual status bit, 34-11

J

Join Numbers, 13-4

K

Kaiser-Bessel Window, 42-6

L

Less Or Equal To 0?, 9-8
 Less Or Equal?, 9-8
 Less Than 0?, 9-8
 Less?, 9-8
 Lexical Class, 9-8
 Line Feed, 6-20
 Linear Fit, 43-4
 Linear Fit Coefficients, 43-4
 List Directory, 11-17
 llo – Local lockout, 34-11
 loc – Go to local, 34-7
 loc – Place Controller in local state, 34-11
 Local Variable, 3-4
 Lock Range, 11-17
 Logarithm Base 2, 4-18
 Logarithm Base 10, 4-18
 Logarithm Base X, 4-18
 Logical Shift, 13-4
 LPM-16 Calibrate, 29-14
 LU Factorization, 45-13

M

Make Alias, 52-13

MakeAddr, 35-10
Mantissa & Exponent, 13-5
Master Slave Config, 29-14
Match Pattern, 6-8
Matrix Condition Number, 45-13
Matrix Norm, 45-13
Matrix Rank, 45-14
Max & Min, 9-9
Mean, 44-10
Measure Frequency, 26-3
Measure Pulse Width or Period, 26-4
Median, 44-10
Median Filter, 41-13
MIO Calibrate (Windows), 29-15
MIO Configure (Windows), 29-16
Mode, 44-10
Moment About Mean, 44-11
Move, 11-17
MSE, 44-11

N

Natural Logarithm, 4-19
Natural Logarithm (Arg +1), 4-19
Network Functions (avg), 40-4
New Directory, 11-17
New File, 11-18
Nonlinear Lev-Mar Fit, 43-5
Normal Distribution, 44-11
Normalize Matrix, 46-4
Normalize Vector, 46-5
Not, 5-4
Not A Notifier, 13-10
Not A Number/Path/Refnum?, 9-9
Not A Path, 11-27
Not A Queue, 13-14
Not A Refnum, 11-27
Not A Rendezvous, 13-16
Not A Semaphore, 13-18
Not And, 5-4
Not Equal To 0?, 9-9
Not Equal?, 9-9
Not Exclusive Or, 5-4
Not Or, 5-4
Number To Boolean Array, 4-11, 5-5
Numeric Integration, 47-2

O

Octal Digit?, 9-10
off – Take controller offline, 34-12
off – Take device offline, 34-7
One Button Dialog Box, 10-8
Open Application Reference, 12-3
Open Automation Refnum, 51-2
Open Config Data, 11-22
Open File, 11-18
Open VI Reference, 12-4
Open/Create/Replace File, 11-7
Or, 5-5
Or Array Elements, 5-5
Out Port (Windows 3.1 and Windows 95), 13-7
Outer Product, 45-14

P

Parks-McClellan, 41-13
PassControl, 35-3
Path Constant, 11-27
Path To Array Of Strings, 11-18, 6-19
Path To String, 11-18, 6-19
Path Type, 11-18
pct – Pass control, 34-8
Peak Detector, 40-5, 47-3
Periodic Random Noise, 38-4
Pick Line & Append, 6-11
Polar To Complex, 4-20
Polynomial Interpolation, 43-5
Power & Frequency Estimate, 40-5
Power Of 2, 4-19
Power Of 10, 4-19
Power Of X, 4-19
Power Spectrum, 39-17
ppc – Parallel poll configure, 34-8
ppc – Parallel poll configure (enable and disable), 34-12
PPC Accept Session, 53-2
PPC Browser, 52-5, 53-2
PPC Close Port, 53-2
PPC End Session, 53-3
PPC Inform Session, 53-3
PPC Open Port, 53-3
PPC Read, 53-4
PPC Start Session, 53-4
PPC Write, 53-5
PPoll, 35-5
PPollConfig, 35-2

PPollUnconfig, 35-5
 ppu – Parallel poll unconfigure, 34-12
 PREFIX Close, 32-2
 PREFIX Error Message, 32-2
 PREFIX Error Query, Error Query (Multiple) and Error Message, 32-3
 PREFIX Initialize and PREFIX Initialize (VXI, Reg-based), 32-3
 PREFIX Message-Based Template and Register-Based Template, 32-4
 PREFIX Register-Based Template, 32-4
 PREFIX Reset, 32-4
 PREFIX Revision Query, 32-5
 PREFIX Self-Test, 32-5
 PREFIX Utility Clean UP Initialize, 32-5
 PREFIX Utility Default Instrument Setup, 32-6
 PREFIX VI Tree, 32-6
 Print Panel, 12-5
 Printable?, 9-10
 Property Node, (Application Control Functions), 12-5,
 Property Node (ActiveX Automation Functions), 51-3
 PseudoInverse Matrix, 45-14
 Pulse Parameters, 40-5
 Pulse Pattern, 38-5
 Pulse Width or Period Meas Config, 27-7

Q

QR Factorization, 45-15
 Quick Scale 1D, 46-6
 Quick Scale 2D, 46-6
 Quit, 12-6

R

Ramp Pattern, 38-5
 Rational Interpolation, 43-5
 RcvRespMsg, 35-8
 Re/Im To Complex, 4-21
 Read Characters From File, 11-7
 Read File, 11-7
 Read from Digital Line, 23-1
 Read from Digital Port, 23-2
 Read From I16 File, 11-13
 Read From SGL File, 11-13
 Read From Spreadsheet File, 11-10
 Read Key (Boolean), 11-22
 Read Key (Double), 11-22
 Read Key (I32), 11-23

Read Key (Path), 11-23
 Read Key (String), 11-23
 Read Key (U32), 11-23
 Read Lines From File, 11-10
 ReadStatus, 35-3
 Real FFT, 39-18
 Receive, 35-3
 ReceiveSetup, 35-9
 Refnum To Path, 11-19, 6-19
 Release Semaphore, 13-18
 Remove Key, 11-24
 Remove Queue Element, 13-14
 Remove Section, 11-24
 Replace Array Element, 7-6
 ResetSys, 35-6
 Reshape Array, 7-6
 Resize Rendezvous, 13-16
 Reverse 1D Array, 7-6
 Reverse String, 6-11
 RMS, 44-12
 Rotate 1D Array, 7-7
 Rotate Left With Carry, 13-5
 Rotate Right With Carry, 13-5
 Rotate String, 6-11
 Rotate, 13-5
 Route Signal, 29-16
 rpp – Conduct parallel poll, 34-12
 rsc – Release or request system control, 34-13
 rsv – Request service and/or set the serial poll status byte, 34-13
 RTSI Control, 29-16

S

Sample Variance, 44-12
 Sawtooth Wave, 38-6
 Scale 1D, 46-7
 Scale 2D, 46-7
 Scaled Time Domain Window, 40-6
 Scaling Constant Tuner, 29-17, 30-9
 Scan From String, 6-11
 Scan String for Tokens, 6-13
 SCXI Cal Constants, 29-17
 SCXI Temperature Scan, 30-11
 Search 1D Array, 7-7
 Secant, 4-16
 Seconds To Date/Time, 10-9
 Seek, 11-19

Select, 9-10
Select & Append, 6-13
Select & Strip, 6-14
Send, 35-3
Send Notification, 13-10
SendCmds, 35-9
SendDataBytes, 35-9
SendIFC, 35-6
SendList, 35-5
SendLLO, 35-7
SendSetup, 35-9
Sequence Structure, 3-2
Serial Port Break, 36-2
Serial Port Init, 36-2
Serial Port Read, 36-2
Serial Port Write, 36-2
Set DAQ Device Information, 29-17
Set Menu Item Info, 12-11
Set Occurrence, 13-20
Set SCXI Information, 29-18
Set Serial Buffer Size, 33-19
SetRWLS, 35-7
SetTimeout, 35-10
sic – Send interface clear, 34-13
Simple Error Handler, 10-11
Sinc, 4-17
Sinc Pattern, 38-7
Sine, 4-17
Sine & Cosine, 4-17
Sine Pattern, 38-7
Sine Wave, 38-8
Solve Complex Linear Equations, 45-15
Solve Linear Equations, 45-16
Sort 1D Array, 7-7
Spectrum Unit Conversion, 40-7
Spline Interpolant, 43-6
Spline Interpolation, 43-7
Split 1D Array, 7-7
Split Number, 13-5
Split String, 6-14
Spreadsheet String To Array, 6-14
Square Wave, 38-8
sre – Unassert or assert remote enable, 34-14
Standard Deviation, 44-12
Stop, 12-6
String Constant, 6-20
String Length, 6-14
String Subset, 6-14

String To Byte Array, 4-11, 6-19
String To IP, 48-3
String To Path, 11-19, 6-20
Strip Path, 11-11
SVD Factorization, 45-17
Swap Bytes, 13-6
Swap Words, 13-6

T

T Distribution, 44-13
Tab, 6-20
Tangent, 4-17
TCP Close Connection, 48-3
TCP Create Listener, 48-3
TCP Listen, 48-2
TCP Open Connection, 48-4
TCP Read, 48-4
TCP Wait on Listener, 48-4
TCP Write, 48-4
Temporary Directory Constant, 11-28
Test Complex Positive Definite, 45-18
Test Positive Definite, 45-18
TestSRQ, 35-7
TestSys, 35-8
Threshold 1D Array, 7-7
Threshold Peak Detector, 40-7, 47-4
Tick Count (ms), 10-9
To Byte Integer, 4-12
To Decimal, 6-17
To Double Precision Complex, 4-12
To Double Precision Float, 4-12
To Engineering, 6-17
To Exponential, 6-17
To Extend Precision Complex, 4-12
To Extended Precision Float, 4-12
To Fractional, 6-17
To Hexadecimal, 6-18
To Long Integer, 4-12
To Lower Case, 6-15
To Octal, 6-18
To Single Precision Complex, 4-13
To Single Precision Float, 4-13
To Unsigned Byte Integer, 4-13
To Unsigned Long Integer, 4-13
To Unsigned Word Integer, 4-13
To Upper Case, 6-15
To Word Integer, 4-13

Trace, 45-18
 Transfer Function, 40-7
 Transpose 2D Array, 7-8
 Triangle Wave, 38-9
 Triangle Window, 42-7
 Trigger, 35-4
 TriggerList, 35-6
 Two Button Dialog Box, 10-9
 Type and Creator, 11-19
 Type Cast, 13-6

U

UDP Close, 49-1
 UDP Open, 49-1
 UDP Read, 49-2
 UDP Write, 49-2
 Unbundle By Name, 8-6
 Unbundle, 8-6
 Unflatten From String, 13-7
 Uniform White Noise, 38-10
 Unit Vector, 46-8
 Unwrap Phase, 39-19
 User Definable Arithmetic Constants, 4-8

V

Variance, 44-13
 VI Library Constant, 11-28
 VISA Assert Trigger, 33-5
 VISA Clear, 33-5
 VISA Close, 33-5
 VISA Disable Event, 33-10
 VISA Discard Events, 33-10
 VISA Enable Event, 33-11
 VISA Find Resource, 33-6
 VISA In8 / In16 / In32, 33-12
 VISA Lock, 33-6
 VISA Map Address, 33-16
 VISA Memory Allocation, 33-13, 33-17
 VISA Memory Free, 33-13, 33-17
 VISA Move In8 / Move In16 / Move In32, 33-14
 VISA Move Out8 / Move Out16 / Move Out32, 33-14
 VISA Open, 33-7
 VISA Out8 / Out16 / Out32, 33-15
 VISA Peek8 / Peek16 / Peek32, 33-17

VISA Poke8 / Poke16 / Poke32, 33-17
 VISA Read, 33-8
 VISA Read STB, 33-9
 VISA Status Description, 33-9
 VISA Unlock, 33-9
 VISA Unmap Address, 33-18
 VISA Wait On Event, 33-11
 VISA Write, 33-9
 Volume Info, 11-20

W

Wait (ms), 10-9
 Wait at Rendezvous, 13-16
 Wait for GPIB RQS, 34-6
 Wait On Notification, 13-10
 Wait On Notification From Multiple, 13-11
 Wait On Occurrence, 13-20
 Wait Until Next ms Multiple, 10-10
 Wait+ (ms), 27-7
 WaitSRQ, 35-8
 While Loop, 3-3
 White Space?, 9-10
 Write Characters To File, 11-11
 Write File, 11-11
 Write Key (Boolean), 11-24
 Write Key (Double), 11-24
 Write Key (I32), 11-25
 Write Key (Path), 11-25
 Write Key (String), 11-25
 Write Key (U32), 11-26
 Write to Digital Line, 23-2
 Write to Digital Port, 23-3
 Write To I16 File, 11-13
 Write To SGL File, 11-14
 Write To Spreadsheet File, 11-12

Y

$Y[i] = \text{Clip}\{X[i]\}$, 39-19
 $Y[i] = X[i-n]$, 39-19

Z

Zero Padder, 39-20