LabWindows/CVI

LabWindows/CVI Standard Libraries Reference Manual

NATIONAL INSTRUMENTS™
The Software is the Instrument™

February 1998 Edition
Part Number 320682D-01

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

© Copyright 1994, 1998 National Instruments Corporation. All rights reserved.

Important Information

Warranty

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

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

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

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

Copyright

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

CVI™, `natinst.com`™, National Instruments™, NI-488™, NI-488.2™, NI-488.2M™, NI-DAQ™, the CVI logo, the National Instruments logo, and The Software is the Instrument™ 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 This Manual xxv
Conventions Used in This Manual xxvii
LabWindows/CVI Documentation Set xxviii
Related Documentation xxviii
Customer Communication xxix

Chapter 1
ANSI C Library

Low-Level I/O Functions 1-2
Standard Language Additions 1-3
Character Processing 1-6
String Processing 1-6
Input/Output Facilities 1-6
File I/O Functions Set errno 1-7
Mathematical Functions 1-7
Time and Date Functions 1-7
    Configuring the DST Rules String 1-8
    Modifying the DST Rules String 1-8
        Suppressing Daylight Savings Time 1-9
        Starting Year in Daylight Savings Time 1-9
Control Functions 1-9
ANSI C Library Function Reference 1-11
    fdopen 1-12

Chapter 2
Formatting and I/O Library

Formatting and I/O Library Function Overview 2-1
    Formatting and I/O Library Function Panels 2-1
    Class and Subclass Descriptions 2-3
    String Manipulation Functions 2-3
    Special Nature of the Formatting and Scanning Functions 2-3
Formatting and I/O Library Function Reference 2-4
    ArrayToFile 2-5
    CloseFile 2-8
    CompareBytes 2-9
    CompareStrings 2-11
    CopyBytes 2-13

CopyString 2-14
FileToArray 2-15
FillBytes 2-18
FindPattern 2-19
Fmt 2-21
FmtFile 2-23
FmtOut 2-24
GetFileInfo 2-25
GetFmtErrNdx 2-26
GetFmtIOError 2-27
GetFmtIOErrorString 2-28
NumFmtdBytes 2-29
OpenFile 2-30
ReadFile 2-32
ReadLine 2-34
Scan 2-36
ScanFile 2-38
ScanIn 2-40
SetFilePtr 2-42
StringLength 2-44
StringLowerCase 2-45
StringUpperCase 2-46
WriteFile 2-47
WriteLine 2-49
Using the Formatting and Scanning Functions 2-50
Introductory Formatting and Scanning Examples 2-50
Formatting Functions 2-51
Formatting Functions—Format String 2-51
Formatting Modifiers 2-54
Fmt, FmtFile, FmtOut—Asterisks (*) Instead of Constants in Format Specifiers 2-61
Fmt, FmtFile, FmtOut—Literals in the Format String 2-61
Scanning Functions 2-62
Scanning Functions—Format String 2-62
Scanning Modifiers 2-66
Scan, ScanFile, ScanIn—Asterisks (*) Instead of Constants in Format Specifiers 2-74
Scan, ScanFile, ScanIn—Literals in the Format String 2-75
Formatting and I/O Library Programming Examples 2-75
Fmt/FmtFile/FmtOut Examples in C 2-76
Integer to String 2-76
Short Integer to String 2-77
Real to String in Floating-Point Notation 2-78

Real to String in Scientific Notation 2-78
Integer and Real to String with Literals 2-79
Two Integers to ASCII File with Error Checking 2-79
Real Array to ASCII File in Columns and
with Comma Separators 2-79
Integer Array to Binary File, Assuming a Fixed Number
of Elements 2-80
Real Array to Binary File, Assuming a Fixed Number
of Elements 2-80
Real Array to Binary File, Assuming a Variable Number
of Elements 2-81
Variable Portion of a Real Array to a Binary File 2-81
Concatenating Two Strings 2-82
Appending to a String 2-83
Creating an Array of Filenames 2-84
Writing a Line That Contains an Integer with Literals
to the Standard Output 2-84
Writing to the Standard Output without a Linefeed
or Carriage Return 2-84
Scan/ScanFile/ScanIn Examples in C 2-85
String to Integer 2-85
String to Short Integer 2-86
String to Real 2-86
String to Integer and Real 2-87
String to String 2-88
String to Integer and String 2-89
String to Real, Skipping over Non-Numeric Characters
in the String 2-89
String to Real, after Finding a Semicolon in the String 2-90
String to Real, after Finding a Substring in the String 2-90
String with Comma-Separated ASCII Numbers to Real Array 2-91
Scanning Strings That Are Not Null-Terminated 2-91
Integer Array to Real Array 2-92
Integer Array to Real Array with Byte Swapping 2-92
Integer Array That Contains 1-Byte Integers to Real Array 2-92
Strings That Contain Binary Integers to Integer Array 2-93
Strings That Contain an IEEE-Format Real Number
to a Real Variable 2-93
ASCII File to Two Integers with Error Checking 2-94
ASCII File with Comma-Separated Numbers to Real Array,
with Number of Elements at Beginning of File 2-94
Binary File to Integer Array, Assuming a Fixed Number
of Elements 2-95

Binary File to Real Array, Assuming a Fixed Number of Elements 2-95
Binary File to Real Array, Assuming a Variable Number of Elements 2-95
Reading an Integer from the Standard Input 2-96
Reading a String from the Standard Input 2-96
Reading a Line from the Standard Input 2-97

Chapter 3
Analysis Library

Analysis Library Function Overview 3-1
Analysis Library Function Panels 3-1
Class and Subclass Descriptions 3-3
Hints for Using Analysis Function Panels 3-4
Reporting Analysis Errors 3-4
Analysis Library Function Reference 3-4
Abs1D 3-5
Add1D 3-6
Add2D 3-7
Clear1D 3-8
Copy1D 3-9
CxAdd 3-10
CxAdd1D 3-11
CxDiv 3-12
CxDiv1D 3-13
CxLinEv1D 3-14
CxMul 3-16
CxMul1D 3-17
CxRecip 3-18
CxSub 3-19
CxSub1D 3-20
Determinant 3-21
Div1D 3-22
Div2D 3-23
DotProduct 3-24
GetAnalysisErrorString 3-25
Histogram 3-26
InvMatrix 3-28
LinEv1D 3-29
LinEv2D 3-30
MatrixMul 3-31
MaxMin1D 3-33

MaxMin2D 3-34
Mean 3-36
Mul1D 3-37
Mul2D 3-38
Neg1D 3-39
Set1D 3-40
Sort 3-41
StdDev 3-42
Sub1D 3-43
Sub2D 3-44
Subset1D 3-45
ToPolar 3-46
ToPolar1D 3-47
ToRect 3-48
ToRect1D 3-49
Transpose 3-50
Error Conditions 3-51

Chapter 4
GPIB/GPIB-488.2 Library

GPIB Library Function Overview 4-1
GPIB Functions Library Function Panels 4-1
Class and Subclass Descriptions 4-4
GPIB Library Concepts 4-5
GPIB Libraries and the GPIB Dynamic Link Library/Device Driver 4-5
Guidelines and Restrictions for Using the GPIB Libraries 4-6
Device and Board Functions 4-6
Automatic Serial Polling 4-7
Autopolling Compatibility 4-8
Hardware Interrupts and Autopolling 4-8
Read and Write Termination 4-8
Timeouts 4-9
Global Variables for the GPIB Library 4-9
Multithreading under Windows 95/NT 4-9
Notification of SRQ and Other GPIB Events under Windows 4-10
Synchronous Callbacks 4-10
Asynchronous Callbacks 4-10
Driver Version Requirements 4-10
GPIB Function Reference 4-11
CloseDev 4-12
CloseInstrDevs 4-13
ibInstallCallback 4-14

iblock 4-17
ibnotify 4-18
ibunlock 4-22
OpenDev 4-23
ThreadIbcnt 4-24
ThreadIbcntl 4-25
ThreadIberr 4-26
ThreadIbsta 4-29

Chapter 5
RS-232 Library

RS-232 Library Function Overview 5-1
RS-232 Library Function Panels 5-1
Class Descriptions 5-2
Using RS-485 5-2
Reporting RS-232 Errors 5-3
XModem File Transfer Functions 5-3
Troubleshooting 5-3
RS-232 Cable Information 5-4
Handshaking 5-6
Software Handshaking 5-7
Hardware Handshaking 5-7
Multithreading under Windows 95/NT 5-8
RS-232 Library Function Reference 5-9
CloseCom 5-10
ComBreak 5-11
ComFromFile 5-12
ComRd 5-14
ComRdByte 5-16
ComRdTerm 5-17
ComSetEscape 5-19
ComToFile 5-21
ComWrt 5-23
ComWrtByte 5-25
FlushInQ 5-26
FlushOutQ 5-27
GetComStat 5-28
GetInQLen 5-30
GetOutQLen 5-31
GetRS232ErrorString 5-32
InstallComCallback 5-33
OpenCom 5-37

OpenComConfig 5-39
ReturnRS232Err 5-42
SetComTime 5-43
SetCTSMode 5-44
SetXMode 5-46
XModemConfig 5-47
XModemReceive 5-49
XModemSend 5-51
Error Conditions 5-52

Chapter 6
DDE Library

DDE Library Function Overview 6-1
DDE Library Function Panels 6-1
DDE Clients and Servers 6-2
DDE Callback Function 6-2
DDE Links 6-4
DDE Library Example Using Microsoft Excel and LabWindows/CVI 6-5
Multithreading under Windows 95/NT 6-6
DDE Library Function Reference 6-6
AdviseDDEDataReady 6-7
BroadcastDDEDataReady 6-10
ClientDDEExecute 6-12
ClientDDERead 6-13
ClientDDEWrite 6-15
ConnectToDDEServer 6-17
DisconnectFromDDEServer 6-20
GetDDEErrorString 6-21
RegisterDDEServer 6-22
ServerDDEWrite 6-25
SetUpDDEHotLink 6-27
SetUpDDEWarmLink 6-28
TerminateDDELink 6-29
UnregisterDDEServer 6-30
Error Conditions 6-31

Chapter 7
TCP Library

TCP Library Function Overview 7-1
TCP Library Function Panels 7-1
TCP Clients and Servers 7-2

TCP Callback Function 7-3
Multithreading under Windows 95/NT 7-4
TCP Library Function Reference 7-4
ClientTCPRead 7-5
ClientTCPWrite 7-6
ConnectToTCPServer 7-7
DisconnectFromTCPServer 7-9
DisconnectTCPClient 7-10
GetHostTCPSocketHandle 7-11
GetTCPErrorString 7-12
GetTCPHostAddr 7-13
GetTCPHostName 7-14
GetTCPPeerAddr 7-15
GetTCPPeerName 7-16
GetTCPSystemErrorString 7-17
RegisterTCPServer 7-18
ServerTCPRead 7-20
ServerTCPWrite 7-21
SetTCPDisconnectMode 7-22
UnregisterTCPServer 7-24
Error Conditions 7-25

Chapter 8
Utility Library

Utility Library Function Overview 8-1
Class Descriptions 8-5
Utility Library Function Reference 8-5
Beep 8-6
Breakpoint 8-7
CheckForDuplicateAppInstance 8-8
CloseCVIRTE 8-10
Cls 8-11
CopyFile 8-12
CVILowLevelSupportDriverLoaded 8-14
CVIRTEHasBeenDetached 8-16
DateStr 8-18
Delay 8-19
DeleteDir 8-20
DeleteFile 8-21
DisableBreakOnLibraryErrors 8-22
DisableInterrupts 8-23
DisableTaskSwitching 8-24

EnableBreakOnLibraryErrors 8-27
EnableInterrupts 8-28
EnableTaskSwitching 8-29
ExecutableHasTerminated 8-30
GetBreakOnLibraryErrors 8-31
GetBreakOnProtectionErrors 8-32
GetCurrentPlatform 8-33
GetCVIVersion 8-34
GetDir 8-35
GetDrive 8-36
GetExternalModuleAddr 8-37
GetFileAttrs 8-40
GetFileDate 8-42
GetFileSize 8-44
GetFileTime 8-46
GetFirstFile 8-48
GetFullPathFromProject 8-51
GetInterruptState 8-53
GetKey 8-54
GetModuleDir 8-56
GetNextFile 8-58
GetPersistentVariable 8-59
GetProjectDir 8-60
GetStdioPort 8-62
GetStdioWindowOptions 8-63
GetStdioWindowPosition 8-64
GetStdioWindowSize 8-65
GetStdioWindowVisibility 8-66
GetSystemDate 8-67
GetSystemTime 8-68
GetWindowDisplaySetting 8-69
InitCVIRTE 8-70
inp 8-72
inpw 8-73
InStandaloneExecutable 8-74
KeyHit 8-75
LaunchExecutable 8-77
LaunchExecutableEx 8-80
LoadExternalModule 8-83
LoadExternalModuleEx 8-88
MakeDir 8-90
MakePathname 8-91
MapPhysicalMemory 8-92

outp 8-95
outpw 8-96
ReadFromPhysicalMemory 8-97
ReadFromPhysicalMemoryEx 8-99
ReleaseExternalModule 8-101
RenameFile 8-103
RetireExecutableHandle 8-105
RoundRealToNearestInteger 8-106
RunExternalModule 8-107
SetBreakOnLibraryErrors 8-109
SetBreakOnProtectionErrors 8-111
SetDir 8-113
SetDrive 8-114
SetFileAttrs 8-115
SetFileDate 8-117
SetFileTime 8-118
SetPersistentVariable 8-120
SetStdioPort 8-121
SetStdioWindowOptions 8-123
SetStdioWindowPosition 8-125
SetStdioWindowSize 8-127
SetStdioWindowVisibility 8-128
SetSystemDate 8-129
SetSystemTime 8-130
SplitPath 8-131
SyncWait 8-133
SystemHelp 8-134
TerminateExecutable 8-137
Timer 8-138
TimeStr 8-139
TruncateRealNumber 8-140
UnloadExternalModule 8-141
UnMapPhysicalMemory 8-142
WriteToPhysicalMemory 8-143
WriteToPhysicalMemoryEx 8-145

Chapter 9
X Property Library

X Property Library Overview 9-1
X Property Library Function Panels 9-1
X Interclient Communication 9-2
Property Handles and Types 9-3

Communicating with Local Applications 9-3
Hidden Window 9-3
Property Callback Functions 9-4
Error Codes 9-4
Using the Library Outside of LabWindows/CVI 9-5
X Property Library Function Reference 9-5
ConnectToXDisplay 9-6
CreateXProperty 9-8
CreateXPropType 9-10
DestroyXProperty 9-13
DestroyXPropType 9-14
DisconnectFromXDisplay 9-15
GetXPropErrorString 9-16
GetXPropertyName 9-17
GetXPropertyType 9-18
GetXPropTypeName 9-19
GetXPropTypeSize 9-20
GetXPropTypeUnit 9-21
GetXWindowPropertyItem 9-23
GetXWindowPropertyValue 9-26
InstallXPropertyCallback 9-30
PutXWindowPropertyItem 9-33
PutXWindowPropertyValue 9-35
RemoveXWindowProperty 9-38
UninstallXPropertyCallback 9-40

Chapter 10
Easy I/O for DAQ Library

Easy I/O for DAQ Library Function Overview 10-1
Advantages of Using the Easy I/O for DAQ Library 10-1
Limitations of Using the Easy I/O for DAQ Library 10-2
Easy I/O for DAQ Library Function Panels 10-2
Class Descriptions 10-3
Device Numbers 10-4
Channel String for Analog Input Functions 10-4
Command Strings 10-5
Channel String for Analog Output Functions 10-6
Valid Counters for the Counter/Timer Functions 10-7
Easy I/O for DAQ Function Reference 10-7
AIAcquireTriggeredWaveforms 10-8
AIAcquireWaveforms 10-14
AICheckAcquisition 10-17

AIClearAcquisition 10-18
AIReadAcquisition 10-19
AISampleChannel 10-21
AISampleChannels 10-23
AIStartAcquisition 10-25
AOClearWaveforms 10-27
AOGenerateWaveforms 10-28
AOUpdateChannel 10-30
AOUpdateChannels 10-31
ContinuousPulseGenConfig 10-32
CounterEventOrTimeConfig 10-35
CounterMeasureFrequency 10-39
CounterRead 10-43
CounterStart 10-44
CounterStop 10-45
DelayedPulseGenConfig 10-46
FrequencyDividerConfig 10-49
GetAILimitsOfChannel 10-53
GetChannelIndices 10-55
GetChannelNameFromIndex 10-57
GetDAQErrorString 10-58
GetNumChannels 10-59
GroupByChannel 10-60
ICounterControl 10-61
PlotLastAIWaveformsPopup 10-63
PulseWidthOrPeriodMeasConfig 10-64
ReadFromDigitalLine 10-67
ReadFromDigitalPort 10-69
SetEasyIOMultitaskingMode 10-71
WriteToDigitalLine 10-72
WriteToDigitalPort 10-74
Error Conditions 10-76

Chapter 11
ActiveX Automation Library

ActiveX Automation Library Function Overview 11-1
Variants and Safe Arrays 11-2
Events are Not Supported 11-2
ActiveX Automation Library Function Panels 11-2
Class Descriptions 11-7
Using Input Variant Parameters 11-8

Using Output Variant Parameters 11-8
Variants Marked as Empty by Retrieval Functions 11-9
Data Types for Variants, Safe Arrays, and Properties 11-9
Handling Dynamic Memory Variants Hold 11-11
ActiveX Automation Library Function Reference 11-11
CA_Array1DToSafeArray 11-12
CA_Array2DToSafeArray 11-13
CA_BSTRGetCString 11-15
CA_BSTRGetCStringBuf 11-16
CA_BSTRGetCStringLen 11-17
CA_CreateObjectByClassId 11-18
CA_CreateObjectByProgId 11-20
CA_CreateObjHandleFromIDispatch 11-22
CA_CStringToBSTR 11-23
CA_DefaultValueVariant 11-24
CA_DiscardObjHandle 11-25
CA_DisplayErrorInfo 11-26
CA_FreeMemory 11-27
CA_FreeUnusedServers 11-28
CA_GetActiveObjectByClassId 11-29
CA_GetActiveObjectByProgId 11-31
CA_GetAutomationErrorString 11-33
CA_GetDispatchFromObjHandle 11-34
CA_GetLocale 11-35
CA_InvokeHelper 11-36
CA_InvokeHelperV 11-41
CA_LoadObjectFromFile 11-42
CA_LoadObjectFromFileByClassId 11-44
CA_LoadObjectFromFileByProgId 11-46
CA_MethodInvoke 11-48
CA_MethodInvokeV 11-50
CA_PropertyGet 11-51
CA_PropertySet 11-53
CA_PropertySetByRef 11-55
CA_PropertySetByRefV 11-57
CA_PropertySetV 11-58
CA_SafeArrayDestroy 11-59
CA_SafeArrayGet1DSize 11-60
CA_SafeArrayGet2DSize 11-61
CA_SafeArrayGetNumDims 11-62
CA_SafeArrayTo1DArray 11-63
CA_SafeArrayTo1DArrayBuf 11-66
CA_SafeArrayTo2DArray 11-69

CA_SafeArrayTo2DArrayBuf 11-72
CA_SetLocale 11-75
CA_VariantBool 11-77
CA_VariantBSTR 11-78
CA_VariantClear 11-79
CA_VariantConvertToType 11-80
CA_VariantCopy 11-82
CA_VariantCurrency 11-83
CA_VariantDate 11-84
CA_VariantDispatch 11-85
CA_VariantDouble 11-86
CA_VariantEmpty 11-87
CA_VariantError 11-88
CA_VariantFloat 11-89
CA_VariantGet1DArray 11-90
CA_VariantGet1DArrayBuf 11-93
CA_VariantGet1DArraySize 11-96
CA_VariantGet2DArray 11-97
CA_VariantGet2DArrayBuf 11-100
CA_VariantGet2DArraySize 11-103
CA_VariantGetArrayNumDims 11-104
CA_VariantGetBool 11-105
CA_VariantGetBoolPtr 11-106
CA_VariantGetBSTR 11-107
CA_VariantGetBSTRPtr 11-108
CA_VariantGetCString 11-109
CA_VariantGetCStringBuf 11-110
CA_VariantGetCStringLen 11-111
CA_VariantGetCurrency 11-112
CA_VariantGetCurrencyPtr 11-113
CA_VariantGetDate 11-114
CA_VariantGetDatePtr 11-115
CA_VariantGetDispatch 11-116
CA_VariantGetDispatchPtr 11-117
CA_VariantGetDouble 11-118
CA_VariantGetDoublePtr 11-119
CA_VariantGetError 11-120
CA_VariantGetErrorPtr 11-121
CA_VariantGetFloat 11-122
CA_VariantGetFloatPtr 11-123
CA_VariantGetInt 11-124
CA_VariantGetIntPtr 11-125
CA_VariantGetIUnknown 11-126

CA_VariantGetIUnknownPtr 11-127
CA_VariantGetLong 11-128
CA_VariantGetLongPtr 11-129
CA_VariantGetObjHandle 11-130
CA_VariantGetSafeArray 11-131
CA_VariantGetSafeArrayPtr 11-133
CA_VariantGetShort 11-135
CA_VariantGetShortPtr 11-136
CA_VariantGetType 11-137
CA_VariantGetUChar 11-138
CA_VariantGetUCharPtr 11-139
CA_VariantGetVariantPtr 11-140
CA_VariantHasArray 11-141
CA_VariantHasBool 11-142
CA_VariantHasBSTR 11-143
CA_VariantHasCString 11-144
CA_VariantHasCurrency 11-145
CA_VariantHasDate 11-146
CA_VariantHasDispatch 11-147
CA_VariantHasDouble 11-148
CA_VariantHasError 11-149
CA_VariantHasFloat 11-150
CA_VariantHasInt 11-151
CA_VariantHasIUnknown 11-152
CA_VariantHasLong 11-153
CA_VariantHasNull 11-154
CA_VariantHasObjHandle 11-155
CA_VariantHasPtr 11-156
CA_VariantHasShort 11-157
CA_VariantHasUChar 11-158
CA_VariantInt 11-159
CA_VariantIsEmpty 11-160
CA_VariantIUnknown 11-161
CA_VariantLong 11-162
CA_VariantNULL 11-163
CA_VariantSet1DArray 11-164
CA_VariantSet2DArray 11-166
CA_VariantSetBool 11-168
CA_VariantSetBoolPtr 11-169
CA_VariantSetBSTR 11-170
CA_VariantSetBSTRPtr 11-171
CA_VariantSetCString 11-172
CA_VariantSetCurrency 11-173

CA_VariantSetCurrencyPtr 11-174
CA_VariantSetDate 11-175
CA_VariantSetDatePtr 11-176
CA_VariantSetDispatch 11-177
CA_VariantSetDispatchPtr 11-178
CA_VariantSetDouble 11-179
CA_VariantSetDoublePtr 11-180
CA_VariantSetEmpty 11-181
CA_VariantSetError 11-182
CA_VariantSetErrorPtr 11-183
CA_VariantSetFloat 11-184
CA_VariantSetFloatPtr 11-185
CA_VariantSetInt 11-186
CA_VariantSetIntPtr 11-187
CA_VariantSetIUnknown 11-188
CA_VariantSetIUnknownPtr 11-189
CA_VariantSetLong 11-190
CA_VariantSetLongPtr 11-191
CA_VariantSetNULL 11-192
CA_VariantSetSafeArray 11-193
CA_VariantSetSafeArrayPtr 11-194
CA_VariantSetShort 11-195
CA_VariantSetShortPtr 11-196
CA_VariantSetUChar 11-197
CA_VariantSetUCharPtr 11-198
CA_VariantSetVariantPtr 11-199
CA_VariantShort 11-200
CA_VariantUChar 11-201
Error Conditions 11-202

Appendix A
Customer Communication

Glossary

Index

Figures

Figure 10-1. One Cycle of a Waveform 10-11
Figure 10-2. Converting a Signal at Periodic Intervals 10-12
Figure 10-3. Resulting Waveform That Resembles Original Waveform 10-12
Figure 10-4. OUT Pin Pulses 10-52

Tables

Table 1-1. ANSI C Standard Library Classes 1-1
Table 1-2. C Locale Information Values 1-3
Table 1-3. p_sign_posn and n_sign_posn Values 1-5
Table 1-4. Error Codes for the system Function under Windows 1-10

Table 2-1. Functions in the Formatting and I/O Library Function Tree 2-2
Table 2-2. Codes That Specify formatcode 2-53
Table 2-3. Formatting Integer Modifiers (%i, %d, %x, %o, %c) 2-55
Table 2-4. Formatting Floating-Point Modifiers (%f) 2-57
Table 2-5. Formatting String Modifiers (%s) 2-59
Table 2-6. Codes That Specify formatcode 2-64
Table 2-7. Scanning Integer Modifiers (%i, %d, %x, %o, %c) 2-67
Table 2-8. Scanning Floating-Point Modifiers (%f) 2-69
Table 2-9. Scanning String Modifiers (%s) 2-71

Table 3-1. Functions in the Analysis Library Function Tree 3-1
Table 3-2. Analysis Library Error Codes 3-51

Table 4-1. Functions in the GPIB/GPIB-488.2 Library Function Tree 4-1

Table 5-1. Functions in the RS-232 Library Function Tree 5-1
Table 5-2. PC Cable Configuration 5-4
Table 5-3. DTE to DCE Cable Configuration 5-5
Table 5-4. PC to DTE Cable Configuration 5-5
Table 5-5. Bit Definitions for the GetComStat Function 5-28
Table 5-6. Valid Event Bits and Descriptions 5-35
Table 5-7. Syntax for Opening Ports 5-37
Table 5-8. Syntax for Opening Ports 5-40
Table 5-9. Valid Mode Values 5-45
Table 5-10. RS-232 Library Error Codes 5-52

Table 6-1. Functions in the DDE Library Function Tree 6-1
Table 6-2. DDE Transaction Types (xType) 6-4
Table 6-3. DDE Library Error Codes 6-31

Table 7-1. Functions in the TCP Library Function Tree 7-1
Table 7-2. TCP Transaction Types (xType) 7-3
Table 7-3. TCP Library Error Codes 7-25

Table 8-1. Functions in the Utility Library Function Tree 8-2
Table 8-2. Functions That Require Low-Level Driver 8-14
Table 8-3. Example Keystrokes and GetKey Return Values 8-54
Table 8-4. Valid windowState Values 8-81

Table 9-1. Functions in the X Property Library Function Tree 9-1
Table 9-2. Predefined Property Types 9-3

Table 10-1. Functions in the Easy I/O for DAQ Library Function Tree 10-2
Table 10-2. Valid Counters 10-7
Table 10-3. Trigger Types 10-10
Table 10-4. Definition of Am9513: Counter+1 10-36
Table 10-5. Valid Internal Timebase Frequencies 10-37
Table 10-6. Adjacent Counters 10-39
Table 10-7. Valid Internal Timebase Frequencies 10-51
Table 10-8. Valid Internal Timebase Frequencies 10-65
Table 10-9. Easy I/O for DAQ Library Error Codes 10-76

Table 11-1. Functions in the ActiveX Automation Library Function Tree 11-3
Table 11-2. Fundamental Data Types for Variants, Safe Arrays, and Properties ... 11-9
Table 11-3. Data Types Modifiers for Variants, Safe Arrays, and Properties 11-10
Table 11-4. operation Parameter Values 11-37
Table 11-5. Return Type Values 11-38
Table 11-6. Parameter Count Values 11-38
Table 11-7. Parameter Types Values 11-39
Table 11-8. Parameter Values 11-39
Table 11-9. Return Values 11-40
Table 11-10. Data Types and Functions to Free Each Element for CA_SafeArrayTo1DArray 11-64
Table 11-11. Data Types and Functions to Free Each CA_SafeArrayTo1DArrayBuf Element 11-67
Table 11-12. Data Types and Functions to Free Each Element for CA_SafeArrayTo2DArray 11-71
Table 11-13. Data Types and Functions to Free Each Element for CA_SafeArrayTo2DArrayBuf 11-74
Table 11-14. Data Types and Functions to Free the Converted Value 11-81
Table 11-15. Data Types and Functions to Free Each Element for CA_VariantGet1DArray 11-91

Table 11-16. Data Types and Functions to Free Each Element for CA_VariantGet1DArrayBuf 11-95
Table 11-17. Data Types and Functions to Free Each Element for CA_VariantGet2DArray 11-99
Table 11-18. Data Types and Functions to Free Each Element for CA_VariantGet2DArrayBuf 11-102
Table 11-19. ActiveX Automation Library Error Codes 11-202

About This Manual

The L*abWindows/CVI Standard Libraries Reference Manual* contains information about the LabWindows/CVI standard libraries: the Graphics Library, the Analysis Library, the Formatting and I/O Library, the GPIB Library, the GPIB-488.2 Library, the RS-232 Library, the Utility Library, and the system libraries. The *LabWindows/CVI Standard Libraries Reference Manual* is intended for LabWindows/CVI users who have already completed the *Getting Started with LabWindows/CVI* tutorial and are familiar with the *LabWindows/CVI User Manual*. To use this manual effectively, you should be familiar with LabWindows/CVI and Windows fundamentals.

Organization of This Manual

The *LabWindows/CVI Standard Libraries Reference Manual* is organized as follows:

- Chapter 1, *ANSI C Library*, describes the ANSI C Standard Library as implemented in LabWindows/CVI.
- Chapter 2, *Formatting and I/O Library*, describes the functions in the LabWindows/CVI Formatting and I/O Library and contains many examples of how to use them. The *Formatting and I/O Library Function Overview* section contains general information about the Formatting and I/O Library functions and panels. The *Formatting and I/O Library Function Reference* section contains an alphabetical list of the function descriptions.
- Chapter 3, *Analysis Library*, describes the functions in the LabWindows/CVI Analysis Library. The *Analysis Library Function Overview* section contains general information about the Analysis Library functions and panels. The *Analysis Library Function Reference* section contains an alphabetical list of the function descriptions.
- Chapter 4, *GPIB/GPIB-488.2 Library*, describes the functions in the LabWindows/CVI GPIB Library. The *GPIB Library Function Overview* section contains general information about the GPIB Library functions and panels, the GPIB DLL, and guidelines and restrictions you should know when using the GPIB Library. The *GPIB Function Reference* section contains an alphabetical list of descriptions for the Device Manager functions, the callback installation functions, and the functions for returning the thread-specific status variables. Refer to

your NI-488.2 or NI-488.2M function reference for detailed descriptions of the NI-488 and NI-488.2 functions.

- Chapter 5, *RS-232 Library*, describes the functions in the LabWindows/CVI RS-232 Library. The *RS-232 Library Function Overview* section contains general information about the RS-232 Library functions and panels. The *RS-232 Library Function Reference* section contains an alphabetical list of function descriptions.
- Chapter 6, *DDE Library*, describes the functions in the LabWindows/CVI DDE (Dynamic Data Exchange) Library. The *DDE Library Function Overview* section contains general information about the DDE Library functions and panels. The *DDE Library Function Reference* section contains an alphabetical list of function descriptions. This library is available for LabWindows/CVI for Windows only.
- Chapter 7, *TCP Library*, describes the functions in the LabWindows/CVI TCP (Transmission Control Protocol) Library. The *TCP Library Function Overview* section contains general information about the TCP Library functions and panels. The *TCP Library Function Reference* section contains an alphabetical list of function descriptions.
- Chapter 8, *Utility Library*, describes the functions in the LabWindows/CVI Utility Library. The Utility Library contains functions that do not fit into any of the other LabWindows/CVI libraries. The *Utility Library Function Overview* section contains general information about the Utility Library functions and panels. The *Utility Library Function Reference* section contains an alphabetical list of function descriptions.
- Chapter 9, *X Property Library*, describes the functions in the Lab/Windows CVI X Property Library. The X Property Library contains functions that read and write properties to and from X Windows. The *X Property Library Overview* section contains general information about the X Property Library functions and panels. The *X Property Library Function Reference* section contains an alphabetical list of function descriptions.
- Chapter 10, *Easy I/O for DAQ Library*, describes the functions in the Easy I/O for DAQ Library. The *Easy I/O for DAQ Library Function Overview* section contains general information about the functions and guidelines and restrictions you should know when using the Easy I/O for DAQ Library. The *Easy I/O for DAQ Function Reference* section contains an alphabetical list of function descriptions.
- Chapter 11, *ActiveX Automation Library*, describes the functions in the ActiveX Automation Library. The *ActiveX Automation Library*

Function Overview section contains general information about the functions as well as guidelines and restrictions you should know when using the ActiveX Automation Library. The *ActiveX Automation Library Function Reference* section contains an alphabetical list of function descriptions.

- Appendix A, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including abbreviations, acronyms, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics 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>.

-

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.

☞

This icon to the left of bold italicized text denotes a note, which alerts you to important information.

⚠

This icon to the left of bold italicized text denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.

bold

Bold text denotes the names of menus, menu items, parameters, or dialog box buttons.

bold italic

Bold italic text denotes a note or caution.

italic	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.
`monospace`	Text in this font denotes text or characters that you should 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, functions, filenames and extensions, and for statements and comments taken from programs.
`monospace italic`	Italic text in this font denotes that you must enter the appropriate words or values in the place of these items.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.
	IEEE 488 and IEEE 488.2 refer to the ANSI/IEEE Standard 488.1-1987 and the ANSI/IEEE Standard 488.2-1992, respectively, which define the GPIB.

LabWindows/CVI Documentation Set

For a detailed discussion of the best way to use the LabWindows/CVI documentation set, refer to the section *Using the LabWindows/CVI Documentation Set* in Chapter 1, *Introduction to LabWindows/CVI* of *Getting Started with LabWindows/CVI*.

Related Documentation

The following documents contain information that you might find helpful as you read this manual:

- ANSI/IEEE Standard 488.1-1987, *IEEE Standard Digital Interface for Programmable Instrumentation.*
- ANSI/IEEE Standard 488.2-1992, *IEEE Standard Codes, Formats, Protocols, and Common Commands.*
- Harbison, Samuel P., and Guy L. Steele, Jr., *C: A Reference Manual*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1995.

- Nye, Adrian. *Xlib Programming Manual*. Sebastopol, CA: O'Reilly & Associates, 1994.
- Gettys, James, and Robert W. Scheifler. *Xlib—C Language X Interface, MIT X Consortium Standard*. Cambridge, MA: X Consortium, 1994.

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

1

ANSI C Library

This chapter describes the ANSI C Standard Library as implemented in LabWindows/CVI, as shown in Table 1-1.

Note ***When you link your executable or DLL with an external compiler, you are using the ANSI C library of the external compiler.***

Table 1-1. ANSI C Standard Library Classes

Class	Header File
Character Handling Character Testing Character Case Mapping	`<ctype.h>`
Date and Time Time Operations Time Conversion Time Formatting	`<time.h>`
Localization	`<locale.h>`
Mathematics Trigonometric Functions Hyperbolic Functions Exp and Log Functions Power Functions	`<math.h>`
Nonlocal Jumping	`<setjmp.h>`
Signal Handling	`<signal.h>`

Table 1-1. ANSI C Standard Library Classes (Continued)

Class	Header File
Input/Output Open/Close Read/Write/Flush Line Input/Output Character Input/Output Formatted Input/Output Buffer Control File Positioning File System Operations Error Handling	`<stdio.h>`
General Utilities String to Arithmetic Expression Random Number Generation Memory Management Searching and Sorting Integer Arithmetic Multibyte Character Sets Program Termination Environment	`<stdlib.h>`
String Handling Byte Operations String Operations String Searching Collation Functions Miscellaneous	`<string.h>`
Low-Level I/O	`<lowlvlio.h>`

Low-Level I/O Functions

Under UNIX, you can use the low-level I/O functions such as `open`, `sopen`, `read`, and `write` from the system library by including system header files in your program. Under Windows, you can use these functions by including `cvi\include\ansi\lowlvlio.h` in your program. LabWindows/CVI provides function panels for these functions.

Standard Language Additions

LabWindows/CVI does not support extended character sets that require more than 8 bits per character. As a result, the wide character type `wchar_t` is identical to the single-byte `char` type. LabWindows/CVI accepts wide character constants that you specify with the `L` prefix (as in `L'ab'`), but only the first character is significant. Furthermore, library functions that use the `wchar_t` type operate only on 8-bit characters.

LabWindows/CVI supports variable argument functions using the ANSI C macros, with one exception: None of the unspecified arguments can have a struct type. As a result, you should never use the macro `va_arg (ap, type)` when `type` is a structure.

Note ***LabWindows/CVI does not warn you about this error.***

Under UNIX, LabWindows/CVI implements only the C locale as defined by the ANSI C standard. The native locale, which is specified by the empty string, `""`, is also the C locale. Table 1-2 shows the locale information values for the C locale.

Table 1-2. C Locale Information Values

Name	Type	C locale Value	Description
`decimal_point`	`char *`	`"."`	Decimal point character for non-monetary values.
`thousands_sep`	`char *`	`""`	Non-monetary digit group separator character or characters.
`grouping`	`char *`	`""`	Non-monetary digit groupings.
`int_curr_symbol`	`char *`	`""`	Three-character international currency symbol, plus the character to separate the international symbol from the monetary quantity.
`currency_symbol`	`char *`	`""`	Local currency symbol for the current locale.
`mon_decimal_point`	`char *`	`""`	Decimal point character for monetary values.
`mon_thousands_sep`	`char *`	`""`	Monetary digit group separator character or characters.

Table 1-2. C Locale Information Values (Continued)

Name	Type	C locale Value	Description
mon_grouping	char *	""	Monetary digit groupings.
positive_sign	char *	""	Sign character or characters for non-negative monetary quantities.
negative_sign	char *	""	Sign character or characters for negative monetary quantities.
int_frac_digits	char	CHAR_MAX	Digits appear to the right of the decimal point for international monetary formats.
frac_digits	char	CHAR_MAX	Digits appear to the right of the decimal point for formats other than international monetary formats.
p_cs_precedes	char	CHAR_MAX	1 if currency_symbol precedes non-negative monetary values; 0 if it follows.
p_sep_by_space	char	CHAR_MAX	1 if a space separates currency_symbol from non-negative monetary values; 0 otherwise.
n_cs_precedes	char	CHAR_MAX	Like p_cs_precedes, for negative values.
n_sep_by_space	char	CHAR_MAX	Like p_sep_by_space, for negative values.
p_sign_posn	char	CHAR_MAX	Positioning of positive_sign for a non-negative monetary quantity, then its currency_symbol.
n_sign_posn	char	CHAR_MAX	Positioning of negative_sign for a negative monetary quantity, then its currency_symbol.

Under Windows, LabWindows/CVI implements the default locale by using the appropriate system settings. Under Windows 95/NT, you can access the system settings by using the Regional Settings applet in the Control Panel. Under Windows 3.1, use the International applet in the Control Panel. Anything not mentioned in this section behaves the same under the default locale as specified in the C locale.

For the `LC_NUMERIC` locale:

- `decimal_point` maps to the value of `sDecimal`.
- `thousands_sep` maps to the value of `sThousand`.

For the `LC_MONETARY` locale:

- `currency_symbol` maps to the value of `sCurrency`.
- `mon_decimal_point` maps to the value of `sDecimal`.
- `mon_thousands_sep` maps to the value of `sThousand`.
- `frac_digits` maps to the value of `iCurrDigits`.
- `int_frac_digits` maps to the value of `iCurrDigits`.
- LabWindows/CVI sets `p_cs_precedes` and `n_cs_precedes` to `1` if `iCurrency` equals `0` or `2`, otherwise it sets them to `0`.
- LabWindows/CVI sets `p_sep_by_space` and `n_sep_by_space` to `0` if `iCurrency` equals `0` or `1`, otherwise they are set to `0`.
- `iNegCurr` determines the value `of p_sign_posn` and `n_sign_posn` as shown in Table 1-3.

Table 1-3. p_sign_posn and n_sign_posn Values

Value of iNegCurr	Value of p_sign_posn and n_sign_posn
0, 4	0
1, 5, 8, 9	1
3, 7, 10	2
6	3
2	4

For the `LC_CTYPE` locale:

- `isalnum` maps to the Windows function `isCharAlphaNumeric`.
- `isalpha` maps to the Windows function `isCharAlpha`.
- `islower` maps to the Windows function `isCharLower`.

- `isupper` maps to the Windows function `isCharUpper`.
- `tolower` maps to the Windows function `AnsiLower`.
- `toupper` maps to the Windows function `AnsiUpper`.

For the `LC_TIME` locale:

- `strftime` uses the following items from the `win.ini` file for the appropriate format specifiers: `sTime`, `iTime`, `s1159`, `s2359`, `iTLZero`, `sShortDate`, and `sLongDate`.
- The names of the weekdays and the names of the months match the language version of LabWindows/CVI.

For the `LC_COLLATE` locale, `strcoll` maps to the Windows function `lstrcmp`.

Because LabWindows/CVI does not support extended character sets that require more than one byte per character, a multibyte character in LabWindows/CVI is actually a single-byte character. Likewise, a multibyte sequence is a sequence of single-byte characters. Because a multibyte character is the same as a wide character, the multibyte conversion functions do little more than return their inputs as outputs.

Character Processing

LabWindows/CVI implements all the ANSI C character processing facilities as both macros and functions. When you run a program in LabWindows/CVI with the debugging level set to Standard or Extended, LabWindows/CVI uses the function versions of the character processing facilities. By using the function versions, LabWindows/CVI can perform run-time user protection on the arguments you pass to the functions.

String Processing

Under UNIX, `strcoll` is equivalent to `strcmp`, and the `LC_COLLATE` locale does not affect its behavior. Under Windows, `strcoll` is equivalent to the Windows function `lstrcmp`. For both platforms, `strxfrm` performs a string copy using `strncpy` and returns the length of its second argument.

Input/Output Facilities

`rename` fails if the target file already exists. Under Windows, `rename` fails if the source and target files are on different disk drives. Under UNIX, `rename` fails if the source and target files are on different file systems.

`fgetpos` and `ftell` set `errno` to `EFILPOS` on error.

File I/O Functions Set errno

The ANSI C file I/O functions and the low-level I/O functions set the `errno` global variable to indicate specific error conditions. The possible values of `errno` are declared in `cvi\include\ansi\errno.h`. A base set of values exists that is common to all platforms, and additional values are specific to particular platforms.

Under Windows 3.1, `errno` gives very limited information. LabWindows/CVI sets `errno` to `EIO` if the operating system returns an error.

Under Windows 95/NT, you can call the Windows SDK `GetLastError` function to obtain system specific information when LabWindows/CVI sets `errno` to one of the following values:

```
EACCES
EBADF
EIO
ENOENT
ENOSPC
```

Mathematical Functions

The macro `HUGE_VAL` defined in the header `math.h` and the macros `FLT_EPSILON`, `FLT_MAX`, `FLT_MIN`, `DBL_EPSILON`, `DBL_MAX`, `DBL_MIN`, `LDBL_EPSILON`, `LDBL_MAX`, and `DBL_MIN` defined in the header `float.h` all refer to variables. Consequently, you cannot use these macros in places where constant expressions are required, such as in global initializations.

Time and Date Functions

`time` returns the number of seconds since Jan. 1, 1990.

`mktime` and `localtime` require time zone information to produce correct results. LabWindows/CVI obtains time zone information from the environment variable `TZ`, if it exists. The value of this variable should have the format `AAA[S]HH[:MM]BBB`, where optional items are in square brackets.

The `AAA` and `BBB` fields specify the names of the standard and daylight savings time (DST) zones, respectively, such as EST for Eastern Standard Time and EDT for Eastern Daylight Time. The optional sign field `S` indicates whether the local time zone is to the west (`+`) or to the east (`-`) of UTC (Greenwich Mean Time). The hour field `HH` and the optional minutes field `:MM` specify the number of hours and minutes from UTC. For example, the string `EST05EDT` specifies the time zone information for the eastern part of the United States.

Configuring the DST Rules String

gmtime, localtime, and mktime make corrections for daylight savings time. By default the DST rules string in cvi\bin\cvimsgs.txt specifies the rules to determine when daylight savings time begins and ends. LabWindows/CVI does, however, honor the Windows 95/NT time zone information when it indicates that your region does not observe daylight savings time. You can override both the DST rules string in cvimsgs.txt and the Windows 95/NT time zone information by entering a modified DST rules string in the LabWindows/CVI configuration options.

In general, a DST rules string contains one or more rules, each beginning with a colon followed by a year in parentheses. The year indicates the first year to which the rule applies. You must put the rules for the more recent years first.

The following is the default value of the DST rules string:

```
":(1986)040102+0:110102-0:(1967)040102-0:110102-0"
```

The default string contains two rules. The first rule applies to years 1986 and later. The second rule applies to years 1967 to 1985.

Each rule consists of a set of descriptors that indicate when to switch between standard and daylight savings time. LabWindows/CVI assumes each year begins in standard time. Each descriptor follows the format MMDDHH+/-Wd. The MM portion identifies a month, where 01 indicates January. The DD portion indicates the day of the month. The month and day together serve as a reference point for the latter portion of the descriptor. The Wd portion is a day of the week, with 0 indicating Sunday, 1 indicating Monday, and so on. The minus sign (–) or plus sign (+) indicates whether the day of the week is the one before (–) or after (+) the month and day the MMDD portion describes. The HH portion indicates the hour of the day. Thus, 040102+0 indicates the first Sunday in April, and 110102-0 indicates the last Sunday in October.

The default DST rules string states that for the years from 1986 to the present, DST begins at 2 a.m. on the first Sunday in April and ends at 2 a.m. on the last Sunday in October. For the years from 1967 to 1985, DST began at 2 a.m. on the last Sunday in March, and ended at 2 a.m. on the last Sunday in October.

Modifying the DST Rules String

If you want to make a change to the DST rules string, you can add an entry to the configuration options for LabWindows/CVI. Set the entry name to DSTRules and set the entry value to the modified DST rules string.

For instructions on how to set configuration options for the LabWindows/CVI development environment, refer to the *How to Set the Configuration Options* section in Chapter 1, *Configuring LabWindows/CVI*, of the *LabWindows/CVI User Manual*. For instructions on how to set configuration options for the LabWindows/CVI Run-time Engine, refer to the

Configuring the Run-Time Engine section in Chapter 7, *Creating and Distributing Standalone Executables and DLLs*, of the *LabWindows/CVI Programmer Reference Manual*.

LabWindows/CVI does not honor the DST rules string in the configuration options of executables or DLLs you build by selecting **Build»Instrument Driver Support Only** in the Project window. The instrument driver support run-time library does not contain code to read the LabWindows/CVI configuration options. To modify the DST rules string, you must do so in `msgrt`*n*`.txt`. Refer to the *Configuring the Run-Time Engine* section in Chapter 7, *Creating and Distributing Standalone Executables and DLLs*, of the *LabWindows/CVI Programmer Reference Manual*.

Suppressing Daylight Savings Time

You can suppress daylight savings time by modifying the DST rules string to the following:

```
":(1990)010102+0:010102-0"
```

Starting Year in Daylight Savings Time

If you start the year in daylight savings time, use the following as the first descriptor in the rule:

```
"010100-0"
```

Control Functions

The `assert` macro that LabWindows/CVI defines does not print diagnostics to the standard error stream when the debugging level is anything other than None. Instead, when the value of its argument evaluates to zero, LabWindows/CVI displays a dialog box with a message that contains the file name, line number, and expression that caused the assert to fail.

Under UNIX, `system` passes the specified command to the Bourne shell (`sh`) as input, as if the current process were performing a `wait(2V)` system call to wait until the shell terminated. LabWindows/CVI does not invoke callbacks while the command executes.

Under Windows, the executable can be an MS-DOS or Windows executable, including `*.exe`, `*.com`, `*.bat`, and `*.pif` files. The function does not return until the command terminates. The LabWindows/CVI runtime engine ignores user keyboard and mouse events until the command exits. Callbacks for asynchronous events, such as idle events, Windows messages, VXI interrupts, `PostDeferredCall` calls, and DAQ events are called while the command is executing.

If you need to execute a command built into `command.com`, such as `copy`, `dir`, and others, you can call `system` with the command `command.com /C DosCommand args`, where `DosCommand` is the shell command you want to execute. Refer to your DOS documentation

for more help with `command.com`. DOS executables (`.exe`, `.com`, and `.bat` files) use the settings in `_default.pif` in your Windows directory when they run. You can change the priority, display options, and more by editing `_default.pif` or by creating another `.pif` file. Refer to your Microsoft Windows documentation for help on creating and editing `.pif` files.

If you pass a `NULL` pointer to the `system` function, LabWindows/CVI returns a nonzero value only if a command processor is available. Under UNIX, if the argument is not a `NULL` pointer, the program returns a zero. Under Windows, if the argument is not a `NULL` pointer, the program returns zero if the program was successfully started; otherwise it returns one of the error codes in Table 1-4.

Table 1-4. Error Codes for the system Function under Windows

Error Code	Description
-1	System was out of memory, executable file was corrupt, or relocations were invalid.
-3	File was not found.
-4	Path was not found.
-6	Attempt was made to dynamically link to a task, or there was a sharing or network protection error occurred.
-7	Library required separate data segments for each task.
-9	There was insufficient memory to start the application.
-11	Windows version was incorrect.
-12	Executable file was invalid. Either it was not a Windows application or the `.exe` image contained an error.
-13	Application was designed for a different operating system.
-14	Application was designed for MS-DOS 4.0.
-15	Type of executable file was unknown.
-16	Attempt made to load a real-mode application developed for an earlier Windows version.
-17	Attempt was made to load a second instance of an executable file containing multiple data segments that were not marked read only.
-20	Attempt was made to load a compressed executable file. You must decompress the file before you can load it.

Table 1-4. Error Codes for the system Function under Windows

Error Code	Description
`-21`	Dynamic-link library (DLL) file was invalid. One of the DLLs required to run this application was corrupt.
`-22`	Application requires Windows 32-bit extensions.
`-23`	Could not find `toolhelp.dll`, or `toolhelp.dll` is corrupted.
`-24`	Could not allocate a `GetProcUserDefinedHandle`.

The LabWindows/CVI environment does not use the argument you pass to `exit`. Under UNIX, standalone executables that LabWindows/CVI creates return the value of the argument you pass to `exit`.

The UNIX version of LabWindows/CVI works with all the ANSI C signals and the signals UNIX supports.

ANSI C Library Function Reference

For ANSI C function descriptions, consult a reference work such as *C: A Reference Manual*, which is listed in the *Related Documentation* section of *About This Manual*. Also, you can use LabWindows/CVI function panel help. The following function description is an extension of the ANSI C function set.

fdopen

```
FILE *fp = fdopen (int fileHandle, char *mode);
```

☞ **Note** ***Only the Windows version of LabWindows/CVI supports*** `fdopen`**.**

Purpose

Creates a buffered I/O stream from a file handle, and returns a pointer to the stream. You can use the return value just as if you had obtained it from `fopen`.

You can obtain a file handle from one of the following functions:

`open` (low-level I/O)
`sopen` (low-level I/O)

Parameters

Input

Name	Type	Description
fileHandle	integer	File handle that `open` or `sopen` returns.
mode	string	Specifies the read/write, binary/text, and append modes.

Return Value

Name	Type	Description
fp	FILE *	Pointer to a buffered I/O file stream.

Return Code

Code	Description
`NULL (0)`	Failure. More specific information is in `errno`.

Parameter Discussion

mode is the same as the **mode** parameter to `fopen`.

You should use a **mode** value that is consistent with the mode in which you originally opened the file. If you use write capabilities that were not enabled when the file handle was originally opened, the call to `fdopen` succeeds, but any attempt to write fails. For instance, if you originally opened the file for reading only, you can pass `"rw"` to `fdopen`, but any call to `fwrite` fails.

2

Formatting and I/O Library

This chapter describes the functions in the LabWindows/CVI Formatting and I/O Library and contains many examples of how to use them. The Formatting and I/O Library contains functions that input and output data to files and manipulate the format of data in a program.

The *Formatting and I/O Library Function Overview* section contains general information about the Formatting and I/O Library functions and panels. Because the Formatting and I/O Library differs in many respects from the other LabWindows/CVI libraries, it is very important to read the overview before you read the other sections of this chapter.

The *Formatting and I/O Library Function Reference* section contains an alphabetical list of function descriptions. This section can help you determine the syntax of the file I/O and string manipulation functions.

The *Using the Formatting and Scanning Functions* section describes in detail this special class of functions. Although the function reference section lists these functions, their versatility and complex nature require a more complete discussion.

The final section, *Formatting and I/O Library Programming Examples*, contains many examples of program code that call Formatting and I/O Library functions. Most of the examples use the formatting and scanning functions.

Formatting and I/O Library Function Overview

This section contains general information necessary for understanding the Formatting and I/O Library functions and panels.

Formatting and I/O Library Function Panels

The Formatting and I/O Library function panels are grouped in the tree structure in Table 2-1 according to the types of operations they perform.

The first- and second-level headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels. Refer to the *Formatting and I/O Library Function Reference* section later in this chapter for more information.

Table 2-1. Functions in the Formatting and I/O Library Function Tree

Class/Panel Name	Function Name
File I/O	
Open File	`OpenFile`
Close File	`CloseFile`
Read from File	`ReadFile`
Write to File	`WriteFile`
Array to File	`ArrayToFile`
File to Array	`FileToArray`
Get File Information	`GetFileInfo`
Set File Pointer	`SetFilePtr`
String Manipulation	
Get String Length	`StringLength`
String to Lowercase	`StringLowerCase`
String to Uppercase	`StringUpperCase`
Fill Bytes	`FillBytes`
Copy Bytes	`CopyBytes`
Copy String	`CopyString`
Compare Bytes	`CompareBytes`
Compare Strings	`CompareStrings`
Find Pattern	`FindPattern`
Read Line	`ReadLine`
Write Line	`WriteLine`
Data Formatting	
Formatting Functions	
Fmt to Memory (Sample Panel)	`Fmt`
Fmt to File (Sample Panel)	`FmtFile`
Fmt to Stdout (Sample Panel)	`FmtOut`
Scanning Functions	
Scan from Mem (Sample Panel)	`Scan`
Scan from File (Sample Panel)	`ScanFile`
Scan from Stdin (Sample Panel)	`ScanIn`
Status Functions	
Get # Formatted Bytes	`NumFmtdBytes`
Get Format Index Error	`GetFmtErrNdx`
Get I/O Error	`GetFmtIOError`
Get I/O Error String	`GetFmtIOErrorString`

Class and Subclass Descriptions

- The File I/O function panels open, close, read, write, and obtain information about files.
- The String Manipulation function panels manipulate strings and character buffers.
- The Data Formatting function panels perform intricate formatting operations with a single function call.
  - Formatting Functions, a subclass of Data Formatting, contains function panels that combine and format one or more source items into a single target item.
  - Scanning Functions, a subclass of Data Formatting, contains function panels that transform a single source item into several target items.
  - Status Functions, a subclass of Data Formatting, contains function panels that return information about the success or failure of a formatting or scanning call.

The online help for each panel contains specific information about operating each function panel.

String Manipulation Functions

The functions in the String Manipulation class perform common operations such as copying one string to another, comparing two strings, or finding the occurrence of a string in a character buffer. These functions are similar in purpose to the standard C string functions.

Special Nature of the Formatting and Scanning Functions

The formatting and scanning functions are different in nature from the other functions in the LabWindows/CVI libraries. With few exceptions, each LabWindows/CVI library function has a fixed number of parameters, and each parameter has a definite data type. Each formatting and scanning function, however, takes a variable number of parameters, and the parameters can be of various data types. This difference is necessary to give the formatting and scanning functions versatility.

For instance, a single `Scan` function call performs disparate operations, such as the following:

- Find the two numeric values in the string `"header: 45, -1.03e-2"` and place the first value in an integer variable and the second value in a real variable.
- Take the elements from an integer array, swap the high and low bytes in each element, and place the resulting values in a real array.

To perform these operations, each formatting and scanning function takes a *format string* as one of its parameters. In effect, a format string is a mini-program that tells the formatting and scanning functions how to transform the input arguments to the output arguments. For conciseness, format strings are constructed using single-character codes. The *Using the Formatting and Scanning Functions* section later in this chapter describes these codes in detail.

You might find the formatting and scanning functions more difficult to learn than other LabWindows/CVI functions. To help you in this learning process, read the discussions in the *Formatting and I/O Library Programming Examples* section at the end of this chapter.

Formatting and I/O Library Function Reference

This section describes each function in the LabWindows/CVI Formatting and I/O Library in alphabetical order.

ArrayToFile

```
int status = ArrayToFile (char *fileName, void *array, int dataType,
                          int numberOfElements, int numberOfGroups,
                          int arrayDataOrder, int fileLayout,
                          int colSepStyle, int fieldWidth,
                          int fileType, int fileAction);
```

Purpose

Saves an array to a file using various formatting options. The function handles creating, opening, writing, and closing the file. You can use `FileToArray` to read back the file into an array.

Parameters

Input

Name	Type	Description
fileName	string	File pathname.
array	void *	Numeric array.
dataType	integer	Array element data type.
numberOfElements	integer	Number of elements in the array.
numberOfGroups	integer	Number of groups in the array.
arrayDataOrder	integer	How to order groups in the file.
fileLayout	integer	Direction to write groups in the file.
colSepStyle	integer	How to separate data on one line.
fieldWidth	integer	Constant width between columns.
fileType	integer	ASCII/binary mode.
fileAction	integer	File pointer reposition location.

Return Value

Name	Type	Description
status	integer	Indicates success/failure.

Return Codes

Code	Description
0	Success.
-1	Error attempting to open file.
-2	Error attempting to close file.
-3	An I/O error occurred.
-4	Invalid **dataType** parameter.
-5	Invalid **numberOfElements** parameter.
-6	Invalid **numberOfGroups** parameter.
-7	Invalid **arrayDataOrder** parameter.
-8	Invalid **fileLayout** parameter.
-9	Invalid **fileType** parameter.
-10	Invalid **colSepStyle** parameter.
-11	Invalid **fieldWidth** parameter.
-12	Invalid **fileAction** parameter.

Parameter Discussion

fileName can be an absolute pathname or a relative file name. If you use a relative file name, `ArrayToFile` creates the file relative to the current working directory.

dataType must be one of the following:

- `VAL_CHAR`
- `VAL_SHORT_INTEGER`
- `VAL_INTEGER`
- `VAL_FLOAT`
- `VAL_DOUBLE`
- `VAL_UNSIGNED_SHORT_INTEGER`
- `VAL_UNSIGNED_INTEGER`
- `VAL_UNSIGNED_CHAR`

If you save the array data in ASCII format, you can divide the array data into groups. `ArrayToFile` can write groups as columns or rows. **numberOfGroups** specifies the number of groups into which to divide the array data. If you do not want to divide your data into groups, use `1`.

If you divide your array data into groups, **arrayDataOrder** specifies how the data is ordered in the array. The two choices are as follows:

- `VAL_GROUPS_TOGETHER`—`ArrayToFile` assumes that the elements of each data group are stored consecutively in the data array.
- `VAL_DATA_MULTIPLEXED`—`ArrayToFile` assumes that the first elements of all data group are stored together, followed by the second elements and so on.

If you save the array data in ASCII format, **fileLayout** specifies how the data appears in the file. The two choices are as follows.

- `VAL_GROUPS_AS_COLUMNS`
- `VAL_GROUPS_AS_ROWS`

If you have only one group, use `VAL_GROUPS_AS_COLUMNS` to write each array element on a separate line.

If you tell `ArrayToFile` to write multiple values on each line, **colSepStyle** specifies how to separate the values. The choices are as follows:

- `VAL_CONST_WIDTH`—constant field width for each column
- `VAL_SEP_BY_COMMA`—values followed by commas, except last value on line
- `VAL_SEP_BY_TAB`—values separated by tabs

If you specify a **colSepStyle** of `VAL_CONST_WIDTH`, **fieldWidth** specifies the width of the columns.

fileType specifies whether to create the file in ASCII or binary format. The choices are as follows:

- `VAL_ASCII`
- `VAL_BINARY`

fileAction specifies the location in the file to begin writing data if the named file already exists. The choices are as follows:

- `VAL_TRUNCATE`—Positions the file pointer to the beginning of the file and deletes its prior contents.
- `VAL_APPEND`—All write operations append data to file.
- `VAL_OPEN_AS_IS`—Positions the file pointer at the beginning of the file but does not affect the prior file contents.

CloseFile

```
int status = CloseFile (int fileHandle);
```

Purpose

Closes the file associated with **fileHandle**. You can obtain a file handle by calling `OpenFile`.

Parameter

Input

Name	Type	Description
fileHandle	integer	File handle.

Return Value

Name	Type	Description
status	integer	Result of the close file operation.

Return Codes

Code	Description
`-1`	Bad file handle.
`0`	Success.

CompareBytes

```
int result = CompareBytes (char *buffer#1, int buffer#1Index, char *buffer#2,
                           int buffer#2Index, int numberOfBytes,
                           int caseSensitive);
```

Purpose

Compares the **numberOfBytes** that start at position **buffer#1Index** of **buffer#1** to the **numberOfBytes** that start at position **buffer#2Index** of **buffer#2**.

Parameters

Input

Name	Type	Description
buffer#1	string	String 1.
buffer#1Index	integer	Starting position in **buffer#1**.
buffer#2	string	String 2.
buffer#2Index	integer	Starting position in **buffer#2**.
numberOfBytes	integer	Number of bytes to compare.
caseSensitive	integer	Case-sensitivity mode.

Return Value

Name	Type	Description
result	integer	Result of the compare operation.

Return Codes

Code	Description
`-1`	Bytes from **buffer#1** less than bytes from **buffer#2**.
`0`	Bytes from **buffer#1** identical to bytes from **buffer#2**.
`1`	Bytes from **buffer#1** greater than bytes from **buffer#2**.

Parameter Discussion

buffer#1Index and **buffer#2Index** are zero-based.

If **caseSensitive** is zero, `CompareBytes` compares alphabetic characters without regard to case. If **caseSensitive** is non-zero, `CompareBytes` considers alphabetic characters equal only if they have the same case.

The function returns an integer value that indicates the lexicographic relationship between the two sets of bytes.

CompareStrings

```
int result = CompareStrings (char *string#1, int string#1Index,
                             char *string#2, int string#2Index,
                             int caseSensitive);
```

Purpose

Compares the null-terminated string that starts at position **string#1Index** of **string#1** to the null-terminated string that starts at position **string#2Index** of **string#2**. **string#1Index** and **string#2Index** are zero-based.

Parameters

Input

Name	Type	Description
string#1	string	String 1.
string#1Index	integer	Starting position in **string#1**.
string#2	string	String 2.
string#2Index	integer	Starting position in **string#2**.
caseSensitive	integer	Case-sensitivity mode.

Return Value

Name	Type	Description
result	integer	Result of the compare operation.

Return Code

Code	Description
`-1`	Bytes from **string#1** less than bytes from **string#2**.
`0`	Bytes from **string#1** identical to bytes from **string#2**.
`1`	Bytes from **string#1** greater than bytes from **string#2**.

Parameter Discussion

If **caseSensitive** is zero, `CompareStrings` compares alphabetic characters without regard to case. If **caseSensitive** is non-zero, `CompareStrings` considers alphabetic characters equal only if they have the same case.

The function returns an integer value that indicates the lexicographic relationship between the two strings.

CopyBytes

```
void CopyBytes  (char targetBuffer[], int targetIndex, char *sourceBuffer,
                         int sourceIndex, int numberOfBytes);
```

Purpose

Copies the **numberOfBytes** bytes that start at position **sourceIndex** of **sourceBuffer** to position **targetIndex** of **targetBuffer**.

Parameters

Input

Name	Type	Description
targetIndex	integer	Starting position in **targetBuffer**.
sourceBuffer	string	Source buffer.
sourceIndex	integer	Starting position in **sourceBuffer**.
numberOfBytes	integer	Number of bytes to copy.

Output

Name	Type	Description
targetBuffer	string	Destination buffer.

Return Value

None.

Parameter Discussion

sourceIndex and **targetIndex** are zero-based.

You can use `CopyBytes` even when **sourceBuffer** and **targetBuffer** overlap.

CopyString

```
void CopyString (char targetString[], int targetIndex, char *sourceString,
                         int sourceIndex, int maximum#Bytes);
```

Purpose

Copies the string that starts at position **sourceIndex** of **sourceString** to position **targetIndex** of **targetString**. `CopyString` stops copying when it encounters an ASCII NUL byte or **maximum#Bytes** bytes. `CopyString` appends an ASCII NUL if no ASCII NUL was copied.

Parameters

Input

Name	Type	Description
targetIndex	integer	Starting position in **targetString**.
sourceString	string	Source buffer.
sourceIndex	integer	Starting position in **sourceString**.
maximum#Bytes	integer	Number of bytes to copy, excluding the ASCII NUL.

Output

Name	Type	Description
targetString	string	Destination buffer.

Return Value

None.

Parameter Discussion

sourceIndex and **targetIndex** are zero-based. If you want to use **maximum#Bytes** to prevent writing beyond the end of **targetString**, make sure to allow room for the ASCII NUL. For example, if **maximum#Bytes** is 40, the destination buffer should contain at least 41 bytes.

If you do not want to specify a maximum number of bytes to copy, use `-1` for **maximum#Bytes**.

You can use `CopyString` when **sourceString** and **targetString** overlap.

☞ **Note** ***The value of* maximum#Bytes *must not exceed one less than the number of bytes in the target variable.***

FileToArray

```
int status = FileToArray (char *fileName, void *array, int dataType,
                          int numberOfElements, int numberOfGroups,
                          int arrayDataOrder, int fileLayout,
                          int fileType);
```

Purpose

Reads data from a file into an array. You can use `FileToArray` with files you create using the `ArrayToFile` function. `FileToArray` handles creating, opening, reading, and closing the file.

Parameters

Input

Name	Type	Description
fileName	string	File pathname.
dataType	integer	Array element data type.
numberOfElements	integer	Number of elements in the array.
numberOfGroups	integer	Number of groups in the array.
arrayDataOrder	integer	How to order groups in the file.
fileLayout	integer	Direction to write groups in the file.
fileType	integer	ASCII/binary mode.

Output

Name	Type	Description
array	void*	Numeric array.

Return Value

Name	Type	Description
status	integer	Indicates success or failure.

Return Codes

Code	Description
0	Success.
-1	Error attempting to open the file.
-2	Error attempting to close the file.
-3	An I/O error occurred.
-4	Invalid **dataType** parameter.
-5	Invalid **numberOfElements** parameter.
-6	Invalid **numberOfGroups** parameter.
-7	Invalid **arrayDataOrder** parameter.
-8	Invalid **fileLayout** parameter.
-9	Invalid **fileType** parameter.

Parameter Discussion

fileName can be an absolute pathname or a relative file name. If you use a relative file name, `FileToArray` locates the file relative to the current working directory.

dataType must be one of the following:

- `VAL_CHAR`
- `VAL_SHORT_INTEGER`
- `VAL_INTEGER`
- `VAL_FLOAT`
- `VAL_DOUBLE`
- `VAL_UNSIGNED_SHORT_INTEGER`
- `VAL_UNSIGNED_INTEGER`
- `VAL_UNSIGNED_CHAR`

numberOfGroups specifies the number of groups into which the data in the file is divided. `FileToArray` can read columns or rows as groups. If you do not want to read your data as groups, use `1`. This parameter applies only if the file type is ASCII.

If you divide your data into groups, **arrayDataOrder** specifies how to store the data in the array. The two choices are as follows:

- `VAL_GROUPS_TOGETHER`—`FileToArray` stores all elements from one data group followed by all elements from the next data group.
- `VAL_DATA_MULTIPLEXED`—`FileToArray` stores the first elements of all data groups consecutively, followed by the second elements from each group and so on.

If the file is in ASCII format, **fileLayout** specifies how the data appears in the file. The two choices are as follows:

- `VAL_GROUPS_AS_COLUMNS`
- `VAL_GROUPS_AS_ROWS`

If there is only one group, `VAL_GROUPS_AS_COLUMNS` specifies that each value in the file is on a separate line.

fileType specifies whether the file is in ASCII or binary format. The choices are as follows.

- `VAL_ASCII`
- `VAL_BINARY`

FillBytes

```
void FillBytes (char buffer[], int startingIndex, int numberOfBytes,
                          int value);
```

Purpose

Sets the **numberOfBytes** bytes that start at position **startingIndex** of **buffer** to the value in the lower byte of **value**. **startingIndex** is zero-based.

Parameters

Input

Name	Type	Description
buffer	string	Destination buffer.
startingIndex	integer	Starting position in **buffer**.
numberOfBytes	integer	Number of bytes to fill.
value	integer	Value to place in bytes.

Return Value

None.

FindPattern

```
int ndx = FindPattern (char *buffer, int startingIndex, int numberOfBytes,
                       char *pattern, int caseSensitive,
                       int startFromRight);
```

Purpose

Searches a character buffer for a pattern of bytes. The string **pattern** specifies the pattern of bytes.

Parameters

Input

Name	Type	Description
buffer	string	Buffer to search.
startingIndex	integer	Starting position in **buffer**.
numberOfBytes	integer	Number of bytes to search.
pattern	string	Pattern to search for.
caseSensitive	integer	Case-sensitivity mode.
startFromRight	integer	Direction of search.

Return Value

Name	Type	Description
ndx	integer	Index in **buffer** where `FindPattern` finds the pattern.

Return Code

Code	Description
`-1`	Pattern not found.

Parameter Discussion

The buffer searched is the set of **numberOfBytes** bytes that starts at position **startingIndex** of **buffer**. If **numberOfBytes** is `-1`, the buffer searched is the set of bytes that starts at position **startingIndex** of **buffer** up to the first ASCII NUL. **startingIndex** is zero-based.

If **caseSensitive** is zero, `FindPattern` compares alphabetic characters without regard to case. If **caseSensitive** is non-zero, `FindPattern` considers alphabetic characters equal only

if they have the same case. If **startFromRight** is zero, `FindPattern` finds the occurrence farthest to the left of the pattern in the buffer. If **startFromRight** is nonzero, `FindPattern` finds the occurrence farthest to the right of the pattern in the buffer.

If `FindPattern` finds the pattern, **pattern** returns the index *relative to the beginning of* **buffer,** where it found the first byte of the pattern. If `FindPattern` does not find the pattern, **pattern** returns `-1`.

The following example returns `4`, which is the index of the second of the three occurrences of `ab` in the string `1ab2ab3ab4`. `FindPattern` skips the first occurrence because **startingIndex** is `3`. Of the two remaining occurrences, `FindPattern` finds the farthest to the left because **startFromRight** is zero:

```
ndx = FindPattern ("1ab2ab3ab4", 3, -1, "AB", 0, 0);
```

On the other hand, the following line returns `7`, which is the index of the last occurrence of `ab` because **startFromRight** is nonzero:

```
ndx = FindPattern ("1ab2ab3ab4", 3, -1, "AB", 0, 1);
```

Fmt

```
int n = Fmt (void *target, char *formatString, source1,...,sourcen);
```

Purpose

Formats the **source1,..., sourcen** arguments according to descriptions in the **formatString** argument.

Parameters

Input

Name	Type	Description
formatString	string	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.
source1,...,sourcen	Types must match **formatString** contents.	

Output

Name	Type	Description
target	Type must match **formatString** contents.	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.

Return Value

Name	Type	Description
n	integer	Number of source format specifiers satisfied.

Return Code

Code	Description
-1	Format string error.

Using This Function

`Fmt` places the result of the formatting into the target argument, which you must pass by reference. The return value indicates how many source format specifiers were satisfied, or `-1` if the format string is invalid. The *Using the Formatting and Scanning Functions* section later in this chapter includes a complete discussion of `Fmt`.

FmtFile

```
int n = FmtFile (int fileHandle, char *formatString, source1,...,sourcen);
```

Purpose

Formats the **source1,..., source*n*** arguments according to descriptions in the **formatString** argument. `FmtFile` writes the result of the formatting into the file that corresponds to the **fileHandle** argument, which you obtain by calling the LabWindows/CVI function `OpenFile`.

Parameters

Input

Name	Type	Description
fileHandle	integer	File handle.
formatString	string	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.
source1,...,source*n*	Types must match **formatString** contents	

Return Value

Name	Type	Description
n	integer	Number of source format specifiers satisfied.

Return Codes

Code	Description
`-1`	Format string error
`-2`	I/O error.

Using This Function

The return value indicates how many source format specifiers were satisfied: `-1` if the format string is invalid, or `-2` if an I/O error occurs. The *Using the Formatting and Scanning Functions* section later in this chapter includes a complete discussion of `FmtFile`.

FmtOut

```
int n = FmtOut (char *formatString, source1,...,sourcen);
```

Purpose

Formats the **source1,...,sourcen** arguments according to descriptions in the **formatString** argument. `FmtOut` writes the result of the formatting to the Standard I/O window.

Parameters

Input

Name	Type	Description
formatString	string	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.
source1,...,sourcen	Types must match **formatString** contents.	

Return Value

Name	Type	Description
n	integer	Number of source format specifiers satisfied.

Return Codes

Code	Description
`-1`	Format string error.
`-2`	I/O error.

Using This Function

The return value indicates how many source format specifiers were satisfied: `-1` if the format string is invalid, or `-2` if an I/O error occurs. The *Using the Formatting and Scanning Functions* section later in this chapter includes a complete discussion of `FmtOut`.

GetFileInfo

```
int status = GetFileInfo (char *fileName, long *fileSize);
```

Purpose

Verifies whether a file exists. Returns an integer value of zero if no file exists and `1` if file exists. **fileSize** is a long variable that contains the file size in bytes or zero if no file exists.

Parameters

Input

Name	Type	Description
fileName	string	Pathname of the file to check.
fileSize	long	File size or zero.

Return Value

Name	Type	Description
status	integer	Indicates whether the file exists.

Return Codes

Code	Description
`1`	File exists.
`0`	File does not exist.
`-1`	Maximum number of files already open.

Example

```
/* Check for presence of file A:\DATA\TEST1.DAT.*/
/* Print its size if file exists or message that states file does not
exist. */
int n;
long size;
n = GetFileInfo("a:\\data\\test1.dat",&size);
if (n == 0)
   FmtOut("File does not exist.");
else
   FmtOut("File size = %i[b4]",size);
```

GetFmtErrNdx

```
int n = GetFmtErrNdx (void);
```

Purpose

Returns the zero-based index into the format string where an error occurred in the last formatting or scanning call.

Parameters

None.

Return Value

Name	Type	Description
n	integer	Position of error in format string.

Return Code

Code	Description
-1	No error.

Using This Function

If the format string of the preceding call contains an error, such as an invalid format or an inappropriate modifier, the return value indicates the position within the format string, beginning with position zero, where the error was found. The function can report only one error per call, even if several errors exist within the string.

Example

```
int i, n;
Scan ("1234", "%s>%d", &i);
n = GetFmtErrNdx ();
/* n will have the value -1, indicating that */
/* no error exists in the format string. */
```

GetFmtIOError

```
int status = GetFmtIOError (void);
```

Purpose

This function returns specific I/O information for the last call to a Formatting and I/O function that performs file I/O. If the last function was successful, `GetFmtIOError` returns zero (no error). If the last function that performs I/O encountered an I/O error, `GetFmtIOError` returns a nonzero value.

Parameters

None.

Return Value

Name	Type	Description
status	integer	Indicates success or failure of last function that performed file I/O.

Return Codes

String	Code	Description
`FmtIONoErr`	0	No error.
`FmtIONoFileErr`	1	File not found.
`FmtIOGenErr`	2	General I/O error.
`FmtIOBadHandleErr`	3	Invalid file handle.
`FmtIOInsuffMemErr`	4	Not enough memory.
`FmtIOFileExistsErr`	5	File already exists.
`FmtIOAccessErr`	6	Permission denied.
`FmtIOInvalArgErr`	7	Invalid argument.
`FmtIOMaxFilesErr`	8	Maximum number of files open.
`FmtIODiskFullErr`	9	Disk is full.
`FmtIONameTooLongErr`	10	Filename is too long.

GetFmtIOErrorString

```
char *message = GetFmtIOErrorString (int errorNum);
```

Purpose

Converts the error number `GetFmtIOError` returns into a meaningful error message.

Parameter

Input

Name	Type	Description
errorNum	integer	Error code you obtain from `GetFmtIOError`.

Return Value

Name	Type	Description
message	string	Explanation of error.

NumFmtdBytes

```
int n = NumFmtdBytes (void);
```

Purpose

Returns the number of bytes formatted or scanned by the previous formatting or scanning call.

Parameters

None.

Return Value

Name	Type	Description
n	integer	Number of bytes formatted or scanned.

Using This Function

If the previous call was a formatting call, `NumFmtdBytes` returns the number of bytes placed into the target. If the previous call was a scanning call, `NumFmtdBytes` returns the number of bytes scanned from the source. The return value is undefined if there are no preceding formatting or scanning calls.

Example

```
double f;   int n;
Scan ("3.1416", "%s>%f", &f);
n = NumFmtdBytes ();
/* n will have the value 6, indicating that */
/* 6 bytes were scanned from the source string. */
```

OpenFile

```
int handle = OpenFile (char *fileName, int read/writeMode, int action,
                           int fileType);
```

Purpose

Opens a file for input and/or output.

Parameters

Input

Name	Type	Description
fileName	string	Pathname.
read/writeMode	integer	Read/write mode.
action	integer	File pointer reposition location.
fileType	integer	ASCII/binary mode.

Return Value

Name	Type	Description
handle	integer	File handle to be used in subsequent ReadFile/WriteFile calls.

Return Code

Code	Description
`-1`	Function failed, unable to open file, or bad argument to function.

Parameter Discussion

fileName is a pathname that specifies the file to open. If the **read/writeMode** argument is write or read/write, `OpenFile` creates the file if it does not already exist. Use `GetFileInfo` to determine whether a file already exists. `OpenFile` creates files with full read and write permissions.

read/writeMode specifies how to open the file:

- `VAL_READ_WRITE`—Open file for reading and writing.
- `VAL_READ_ONLY`—Open file for reading only.
- `VAL_WRITE_ONLY`—Open file for writing only.

action specifies whether to delete the old contents of the file and whether to force the file pointer to the end of the file before each write operation. **action** is meaningful only if **read/writeMode** is `VAL_READ_WRITE` or `VAL_WRITE_ONLY`. After you perform read operations, the file pointer points to the byte that follows the last byte read. **action** values are as follows:

- `VAL_TRUNCATE`—Truncate file; delete its old contents and position the file pointer at the beginning of the file.
- `VAL_APPEND`—Do not truncate file; append all write operations to end of file.
- `VAL_OPEN_AS_IS`—Do not truncate file; position the file pointer at the beginning of the file.

fileType specifies whether to treat the file as ASCII or binary. When you perform I/O on a file in binary mode, carriage returns (`CR`) and linefeeds (`LF`) receive no special treatment. When you open the file in ASCII mode, `CR LF` combination translates to `LF` when reading, and `LF` translates to `CR LF` when writing. **fileType** values are as follows:

- `VAL_BINARY`—binary
- `VAL_ASCII`—ASCII

ReadFile

```
int n = ReadFile (int fileHandle, char buffer[], int count);
```

Purpose

Reads up to **count** bytes of data from a file or the Standard Input into **buffer**. Reading starts at the current position of the file pointer. When `ReadFile` completes, the file pointer points to the next unread character in the file.

Parameters

Input

Name	Type	Description
fileHandle	integer	File handle.
count	integer	Number of bytes to read.

Output

Name	Type	Description
buffer	string	Input buffer.

Return Value

Name	Type	Description
n	integer	Number of bytes read.

Return Codes

Code	Description
`-1`	Error, possibly a bad handle.
`0`	Tried to read past end-of-file.

Parameter Discussion

To read from a file, first call `OpenFile` to obtain a **fileHandle**. To read from the Standard Input, pass `0` for **fileHandle**. **buffer** is the buffer into which you read data. You must allocate space for this buffer before you call this function. **count** specifies the number of bytes to read. **count** must not be greater than **buffer** size.

Using This Function

The return value can be less than the number of bytes requested if `ReadFile` reaches the end of the file before the byte count is satisfied. If you open the file in ASCII mode, `ReadFile` counts each `CR LF` combination read as one character, because the pair is translated into `LF` when `ReadFile` stores it in the buffer.

Note `ReadFile` ***does not terminate the buffer with an ASCII NUL.***

ReadLine

```
int n = ReadLine (int fileHandle, char lineBuffer[],int maximum#Bytes);
```

Purpose

Reads bytes from a file until it encounters a linefeed.

Parameters

Input

Name	Type	Description
fileHandle	integer	File handle.
maximum#Bytes	integer	Maximum number of bytes to read into line, excluding the ASCII NUL.

Output

Name	Type	Description
lineBuffer	string	Input buffer.

Return Value

Name	Type	Description
n	integer	Number of bytes read, excluding linefeed.

Return Codes

Code	Description
`-2`	End of file.
`-1`	I/O error.

Parameter Discussion

`ReadLine` places up to **maximum#Bytes** bytes, excluding `LF`, into **lineBuffer** and appends an ASCII NUL to **lineBuffer**. If there are more than **maximum#Bytes** bytes before `LF`, `ReadLine` discards the extra bytes.

Call `OpenFile` to obtain **fileHandle**. You should open the file in ASCII mode so that `ReadLine` treats a `CR LF` combination as `LF`. If **fileHandle** is zero, `ReadLine` reads the line from the Standard Input.

lineBuffer is a character buffer that should be large enough to contain **maximum#Bytes** bytes plus an ASCII NUL.

`ReadLine` returns the number of bytes read from the file, including discarded bytes, but excluding `LF`. Hence, the return value exceeds **maximum#Bytes** if bytes are discarded.

If `ReadLine` reads no bytes because it has already reached the end of the file, it returns `-2`. If an I/O error occurs, `ReadLine` returns `-1`.

Scan

```
int n = Scan (void *source, char *formatString, targetptr1,...,targetptrn);
```

Purpose

Scans a single source item in memory and breaks it into component parts according to format specifiers found in a **formatString**. `Scan` then places the components into the target parameters.

Parameters

Input

Name	Type	Description
source	string	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.
formatString	Types must match **formatString** contents.	

Output

Name	Type	Description
targetptr1,..., targetptr*n*	Types must match **formatString** contents.	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.

Return Value

Name	Type	Description
n	integer	Number of target format specifiers satisfied.

Return Code

Code	Description
`-1`	Format string error.

Using This Function

The return value indicates how many target format specifiers were satisfied or `-1` if the format string is invalid. The *Using the Formatting and Scanning Functions* section later in this chapter includes a complete discussion of `Scan`.

ScanFile

```
int n = ScanFile (int fileHandle, char *formatString,
                          targetptr1,...,targetptrn);
```

Purpose

Performs the same basic operation as the `Scan` function, except that `ScanFile` reads the source data from the file you specify in the **fileHandle** argument. You obtain **fileHandle** by calling the LabWindows/CVI function `OpenFile`.

Parameters

Input

Name	Type	Description
fileHandle	integer	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.
formatString	Types must match **formatString** contents.	

Output

Name	Type	Description
targetptr1,..., targetptr*n*	Types must match **formatString** contents.	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.

Return Value

Name	Type	Description
n	integer	Number of target format specifiers satisfied.

Return Codes

Code	Description
`-1`	Format string error.
`-2`	I/O error.

Using This Function

The amount of data `ScanFile` reads from the file depends on the amount needed to fulfill the target format specifiers in the format string. The return value indicates how many target format specifiers were satisfied; `-1` if the format string is invalid or `-2` if an I/O error occurs. The *Using the Formatting and Scanning Functions* section later in this chapter includes a complete discussion of `ScanFile`.

ScanIn

```
int n = ScanIn (char *formatString, targetptr1,...,targetptrn);
```

Purpose

Performs the same basic operation as `ScanFile`, except that `ScanIn` obtains the source data from the Standard Input.

Parameters

Input

Name	Type	Description
formatString	string	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.

Output

Name	Type	Description
targetptr1,..., targetptr*n*	Types must match **formatString** contents.	Refer to the *Using the Formatting and Scanning Functions* section later in this chapter.

Return Value

Name	Type	Description
n	integer	Number of target format specifiers satisfied.

Return Codes

Code	Description
`-1`	Format string error.
`-2`	I/O error.

Using This Function

`ScanIn` requires no argument for the source item. The return value indicates how many target format specifiers were satisfied; `-1` if the format string is invalid or `-2` if an I/O error occurs. The *Using the Formatting and Scanning Functions* section later in this chapter includes a complete discussion of `ScanIn`.

SetFilePtr

```
long position = SetFilePtr (int fileHandle, long offset, int origin);
```

Purpose

Moves the file pointer for the file specified by **fileHandle** to a location that is **offset** bytes from **origin**. Returns the offset of the new file pointer position from the beginning of the file.

Parameters

Input

Name	Type	Description
fileHandle	integer	File handle you obtain from `OpenFile`.
offset	long integer	Number of bytes from origin to position of file pointer.

Output

Name	Type	Description
origin	integer	Position in file from which to base offset.

Return Value

Name	Type	Description
position	long integer	Offset of the new file pointer position from the beginning of the file.

Return Code

Code	Description
`-1`	An invalid file handle, an invalid origin value, or an offset value that is before the beginning of the file.

Parameter Discussion

The valid values of **origin** are as follows:

- `0` = beginning of file
- `1` = current position of file pointer
- `2` = end of file

Using This Function

You can use `SetFilePtr` to obtain the file size by setting offset to `0` and origin to `2`. In this case, the return value indicates the file size, and the file pointer points to at the end of the file.

You can position the file pointer beyond the end of the file. Intermediate bytes, bytes between the old end of file and the new end of file, contain values that might vary. An attempt to position the file pointer before the beginning of the file causes the function to return an error.

If the file is a device that does not support random access, such as the Standard Input, the function returns a value that might vary.

Example

```
/* Open or create the file c:\TEST.DAT, move 10 bytes into the file,
   and write a string to the file. */
/* Note: Use \\ in pathname in C instead of \. */
int handle, result;
long position;
handle = OpenFile("c:\\TEST.DAT", 0, 2, 1);
if (handle == -1){
   FmtOut("error opening file");
   exit(1);
}
position = SetFilePtr(handle, 10L, 0);
if (position == 10){
   result = WriteFile(handle, "Hello, World!", 13);
   if (result == -1)
      FmtOut("error writing to file");
}
else
   FmtOut("error positioning file pointer");
CloseFile(handle);
```

StringLength

```
int n = StringLength (char *string);
```

Purpose

Returns the number of bytes in the **string** before the first ASCII NUL.

Parameter

Input

Name	Type	Description
string	string	Null-terminated string.

Return Value

Name	Type	Description
n	integer	Number of bytes in **string** before ASCII NUL.

Example

```
char s[100];
int nbytes;
nbytes = StringLength (s);
```

StringLowerCase

```
void StringLowerCase (char string[]);
```

Purpose

Converts all uppercase alphabetic characters in the null-terminated **string** to lowercase.

Parameter

Input/Output

Name	Type	Description
string	string	String to convert to lowercase.

Return Value

None.

StringUpperCase

```
void StringUpperCase (char string[]);
```

Purpose

Converts all lowercase alphabetic characters in the null-terminated **string** to uppercase.

Parameter

Input/Output

Name	Type	Description
string	string	String to convert to uppercase.

Return Value

None.

WriteFile

```
int n = WriteFile (int fileHandle, char *buffer, unsigned int count);
```

Purpose

Writes up to **count** bytes of data from **buffer** to a file or to the Standard Output. `WriteFile` starts writing at the current position of the file pointer. When `WriteFile` completes, it increments the file pointer by the number of bytes written.

Parameters

Input

Name	Type	Description
fileHandle	integer	File handle.
buffer	string	Data buffer.
count	integer	Number of bytes to write.

Return Value

Name	Type	Description
n	integer	Number of bytes written to the file.

Return Code

Code	Description
`-1`	Error. An error can indicate a bad file handle, an attempt to access a protected file, an attempt to write to a file opened as read only, or no more space left on disk.

Parameter Discussion

To write data to a file, first call `OpenFile` to obtain a **fileHandle**. To write to the Standard Output, pass `1` for **fileHandle**.

buffer is the buffer from which to write data.

count specifies the number of bytes to write. The **count** parameter overrides the buffer size in determining the number of bytes to write. Buffers that contain embedded ASCII NUL bytes are written in full. **count** must not be greater than **buffer** size.

Using This Function

For files you open in ASCII mode, `WriteFile` replaces each `LF` character with a `CR LF` combination in the output. The return value does not include the `CR` characters `WriteFile` inserts before the `LF` characters.

WriteLine

```
int n = WriteLine (int fileHandle, char *lineBuffer,int numberOfBytes);
```

Purpose

Writes **numberOfBytes** bytes from **lineBuffer** to a file and then writes a linefeed to the file.

Parameters

Input

Name	Type	Description
fileHandle	integer	File handle.
lineBuffer	string	Data buffer.
numberOfBytes	integer	Number of bytes to write.

Return Value

Name	Type	Description
n	integer	Number of bytes written, including linefeed.

Return Code

Code	Description
`-1`	I/O error.

Parameter Discussion

If **numberOfBytes** is `-1`, `WriteLine` writes only the bytes in **lineBuffer** before the first ASCII NUL, followed by `LF`.

Call `OpenFile` to obtain a **fileHandle**. You should open the file in ASCII mode so that `WriteLine` writes `CR` before `LF`. If **fileHandle** is `1`, `WriteLine` writes the line to the Standard Output.

Using This Function

`WriteLine` returns the number of bytes written to the file, excluding `LF`. If an I/O error occurs, `WriteLine` returns `-1`.

Using the Formatting and Scanning Functions

You use data formatting functions to translate or reformat data items into other forms. Typical uses might be to translate between data stored on external files and the internal forms the program can manipulate, or to reformat a foreign binary representation into one on which the program can operate.

Three subclasses of data formatting functions exist in the LabWindows/CVI Formatting and I/O Library:

- Formatting functions
- Scanning functions
- Status functions

You use formatting functions to combine and format one or more source items into a single target item, and you use scanning functions to break apart a single source item into several target items. The status functions return information regarding the success or failure of the formatting or scanning functions.

Introductory Formatting and Scanning Examples

To introduce you to the formatting and scanning functions, consider the following examples.

Convert the integer value 23 to its ASCII representation and place the contents in a string variable:

```
char a[5];
int b,n;
b = 23;
n = Fmt (a, "%s<%i", b);
```

After the `Fmt` call, `a` contains the string `23`.

In this example, `a` is the target argument, `b` is the source argument, and the string `%s<%i` is the format string. The `Fmt` call uses the format string to determine how to convert the source argument into the target argument.

With `Scan`, you can convert the string `23` to an integer:

```
char *a;
a = "23";
n = Scan (a$, "%s>%i", b%);
```

After the `Scan` call, `b` = `23`.

In this example, `a` is the source argument, `b` is the target argument, and `%s>%i` is the format string. In both the formatting and the scanning functions, the format string defines the variable types of the source and target arguments and the method by which the source arguments are transformed into the target arguments.

Formatting Functions

The following information is a brief description of the three formatting functions:

- `n = Fmt (target, formatstring, source1, ..., source`*n*`);`

  The `Fmt` function formats the `source1, ..., source`*n* arguments according to descriptions in the `formatstring` argument. `Fmt` places the result of the formatting into the `target` argument.

- `n = FmtFile (handle, formatstring, source1, ..., source`*n*`);`

  The `FmtFile` function formats the `source1, ..., source`*n* arguments according to descriptions in the `formatstring` argument. `FmtFile` writes the result of the formatting into the file corresponding to the `handle` argument.

- `n = FmtOut (formatstring, source1, ..., source`*n*`);`

  The `FmtOut` function formats the `source1, ..., source`*n* arguments according to descriptions in the `formatstring` argument. `FmtOut` writes the result of the formatting to the Standard Output.

Each of these formatting functions returns the number of `source` format specifiers satisfied. If an error exists in the format string, the functions return `-1`.

The formatting functions format and combine multiple source items into a single target item. The only difference in the workings of the three functions is the location of the target data. For `Fmt`, the target is a data item in memory that you pass to the function by reference. You must pass the target parameter for `Fmt` by reference. For `FmtFile`, the target is a file whose handle you pass as the first argument. Call `OpenFile` to obtain a file handle. For `FmtOut`, the target is the Standard Output, typically the display. `FmtOut` omits the target argument present in the other two functions. Except for these differences, the following descriptions apply to all the formatting functions.

Formatting Functions—Format String

Consider the following formatting function:

`n = Fmt(target, formatstring, source1, ..., source`*n*`);`

where `formatstring` contains the information to transform the source arguments to the target argument.

Format strings for all the formatting functions are of the form:

```
"target_spec < source_specs_and_literals"
```

where `target_spec` is a format specifier that describes the nature of the target data item, and `source_specs_and_literals` is a sequence of format specifiers and literal characters that indicate how to combine the source material into the target.

Examples of format strings for the formatting functions are as follows:

```
"%s < RANGE %i"
"%s < %s; %i"
```

The character '<' serves as a visual reminder of the direction of the data transformation from the sources to the target and separates the single target format specifier from the source format specifiers and literals. You can omit the target format specifier, in which case the functions assume a `%s` string format. If you omit the target format specifier, you can omit the '<' character or retain it for clarity.

Notice that the target format specifier is located to the left of the '<' symbol, just as the target parameter is located to the left of the format string. Likewise, the source format specifiers are located to the right of the '<' symbol, just as the source parameters are located to the right of the format string.

Format specifiers describe the inputs and outputs of data transformations. Each format specifier has the following form:

```
%   [ rep ]   formatcode   [[ modifiers ]]
```

The character `'%'` introduces all format specifiers. `rep` indicates how many times the format repeats with respect to the arguments. `formatcode` is a code character that indicates the nature of the data items you want to format. `modifiers` is an optional bracket-enclosed sequence of codes that further describe the data format.

Examples of format specifiers are as follows:

```
%s   %100f   %i[b2u]
```

Note `rep` ***is not allowed when*** `formatcode` ***is*** `s` ***(string).***

`formatcode` is specified with one of the following codes in Table 2-2.

Table 2-2. Codes That Specify formatcode

formatcode	Meaning	Description
`s`	String	As a source or target specifier, this indicates that the corresponding parameter is a character string. As a target specifier, this can mean that numeric source parameters become converted into an ASCII form for inclusion in the target string. Refer to the individual numeric formats, such as `%i` and `%f`, for details of these conversions. `s` is the default if no target specifier exists. The functions do not work on arrays of strings. For example, `%10s` is not a valid format string. **Note**: When the functions fill in a target string, they always place an ASCII NUL in the string after the last byte.
`i`	Integer	This source or target specifier indicates that the corresponding parameter is an integer or, if `rep` is present, an integer array. The function performs conversions to ASCII digits when converting to or from the string format `%s`. A modifier is available to specify the radix to use in such a conversion. The default is decimal.
`x`	Integer (hexadecimal)	This source or target specifier indicates that the corresponding parameter is an integer or, if `rep` is present, an integer array. The function performs conversions to ASCII hexadecimal digits (`0123456789abcdef`) when converting to or from the string format `%s`.
`o`	Integer (octal)	This source or target specifier indicates that the corresponding parameter is an integer or, if `rep` is present, an integer array. The function performs conversions to ASCII octal digits (`01234567`) when converting to or from the string format `%s`.
`d`	Integer (decimal)	This format specifier is identical to `%i` and is included for compatibility with the C `printf` family of functions.

Table 2-2. Codes That Specify formatcode (Continued)

formatcode	Meaning	Description
`f`	Real number	This source or target specifier indicates that the corresponding parameter is a real number or, if `rep` is present, a real array. The functions perform conversions to ASCII when converting to or from the string format `%s`.
`c`	Character	This source or target specifier indicates that the corresponding parameter is an integer with one significant byte or, if `rep` is present, an array of 1-byte integers. The functions do *not* perform conversion to ASCII when converting to or from the string format `%s`. The functions copy *directly* to or from the string.

Formatting Modifiers

`modifiers` are optional codes you use to describe the nature of the source or target data. If you use them, you must enclose the modifiers in square brackets and place them immediately after the format code they modify. If one format specifier requires more than one modifier, enclose all modifiers in the same set of brackets.

A different set of modifiers exists for each possible format specifier as shown in Table 2-3, Table 2-4, and Table 2-5.

Table 2-3. Formatting Integer Modifiers (%i, %d, %x, %o, %c)

Modifier	Meaning	Description
b*n*	Specify length	The b integer modifier specifies the length of the integer argument, or the length of an individual integer array element, in bytes. The default length is 4 bytes; therefore, simple 4-byte integers do not need this modifier. The modifier b2 represents short integers. The modifier b1 represents single-byte integers.
i*n*	Specify array offset	The i integer modifier specifies an offset within an integer array argument. It indicates the location within the array where processing begins. *n* is the zero-based index of the first element to process. Thus, %10d[i2] applied to a source integer array processes the 10 integer values from the third–12th elements of the array. The i modifier is valid only if rep is present. If you use the i modifier with the z modifier, *n* is in terms of bytes.
z	Treat string as integer array	The z integer modifier indicates that the data type of the corresponding argument is a string. Nevertheless, the functions treat the data in the string as an integer array. The z modifier is valid only if rep is present.
r*n*	Specify radix	The r integer modifier specifies the radix of the integer argument, which is important if the functions convert the integer into string format. Legal radixes are 8 (octal), 10 (decimal, the default), 16 (hexadecimal), and 256 (a special radix that represents single 8-bit ASCII characters).
w*n*	Specify string size	The w integer modifier specifies the exact number of bytes in which to store a string representation of the integer argument, in the event that the functions convert the integer to a string format. You can enter any non-negative value here. If *n* is less than the number of digits required to represent the integer, the functions insert an asterisk (*) into the string to signify an overflow. The default for *n* is zero, which indicates that the integer can occupy whatever space is necessary.

Table 2-3. Formatting Integer Modifiers (%i, %d, %x, %o, %c) (Continued)

Modifier	Meaning	Description
p*c*	Specify padding	The p integer modifier specifies a padding character *c*, which fills the space to the left of an integer in the event it does not require the entire width you specify with the w*n* modifier. The default padding character is a blank.
s	Specify as two's complement	The s integer modifier indicates that the functions consider the integer argument a signed two's complement number. This is the default interpretation of integers, so the functions do not require the s modifier.
u	Specify as unsigned	The u integer modifier indicates that the functions consider the integer an unsigned integer.
o*nnnn*	Specify byte ordering	You use the o integer modifier to describe the byte ordering of raw data so that LabWindows/CVI can map it to the byte order appropriate for the Intel (PC) or Motorola (SPARCstation) architecture. The number of *n*'s must be equal to the byte size of the integer argument as specified by the b*n* modifier, which must precede the o modifier. In the case of a four-byte integer, o0123 indicates that the bytes are in ascending order of precedence (Intel style), and o3210 indicates that the bytes are in descending order of precedence (Motorola style).

When using the Fmt function to transfer raw instrument data to or from a C integer or integer array, you must use the o modifier on the buffer that contains the raw instrument data. Do not use the o modifier on the buffer that contains the C integer or integer array. LabWindows/CVI determines the byte ordering of the buffer without the o modifier based on the architecture on which your program is running.

For example, if your GPIB instrument sends 2-byte binary data in Intel byte order, your code should appear as follows:

```
short int instr_buf[100];
short int prog_buf[100];
status = ibrd (ud, instr_buf, 200);
Fmt (prog_buf, "%100d<%100d[b2o01]", instr_buf);
```

If, instead, your GPIB instrument sends two-byte binary data in Motorola byte order, Fmt should appear as follows:

```
Fmt (prog_buf, "%100d<%100d[b2o10]", prog_buf);
```

In either case, you use the `o` modifier only on the buffer that contains the raw data from the instrument (`instr_buf`). LabWindows/CVI ensures that the program buffer (`prog_buf`) is in the proper byte order for the host processor.

Note ***When you use both the*** `bn` ***and*** `on` ***modifiers on an integer specifier, the*** `bn` ***modifier must be first.***

Table 2-4. Formatting Floating-Point Modifiers (%f)

Modifier	Meaning	Description
`b`*n*	Specify length	The `b` floating-point modifier specifies the length of the floating-point argument, or the length of an individual array element, in bytes. The default length is 8 bytes; therefore, double-precision values do not need this modifier. Single-precision, floating-point values are indicated by `b4`. The only valid values for *n* are `8` and `4`.
`i`*n*	Specify array offset	The `i` floating-point modifier specifies an offset within a floating-point array argument. It indicates the location within the array where processing begins. *n* is the zero-based index of the first element to process. Thus, `%10f[i2]` applied to a source floating-point array processes the 10 floating-point values from the third–12th elements of the array. The `i` modifier is valid only if `rep` is present. If you use the `i` modifier with the `z` modifier, *n* is in terms of bytes.
`z`	Treat string as floating-point array	The `z` floating-point modifier indicates that the data type of the corresponding argument is a string. Nevertheless, the functions treat the data in the string as a floating-point array. The `z` modifier is valid only if `rep` is present.
`w`*n*	Specify string size	The `w` floating-point modifier specifies the exact number of bytes in which to store a string representation of the floating-point argument, in the event that the functions convert the value to a string format. You can enter any non-negative value here. If *n* is less than the number of digits required to represent the floating-point number, the functions insert an asterisk (`*`) into the string to signify an overflow. The default for *n* is zero, which indicates that the value can occupy whatever space is necessary.

Table 2-4. Formatting Floating-Point Modifiers (%f) (Continued)

Modifier	Meaning	Description
p*n*	Specify precision	The p floating-point modifier specifies the number of digits to the right of the decimal point in a string representation of the floating-point number. You can lose significant digits by attempting to conform to the precision specification. If you omit the p*n* modifier, the default value is p6.
e*n*	Specify as scientific notation	The e floating-point modifier instructs the functions to convert a value to string format in scientific notation. If you omit the e*n* modifier, the functions use the floating-point notation. *n* is optional and specifies the number of digits in the exponent. For example, %f[e2] formats 10.0 as 1.0e+01. If you omit *n*, the functions use a default of three. **Note**: The functions can represent the value in scientific notation even when the e modifier is absent. This occurs when the absolute value of the argument is greater than 1.0e40 or less than 1.0e-40, or when the absolute value of the argument is greater than 1.0e20 or less than 1.0e-4 and neither the p modifier nor the w modifier is present.
f	Specify as floating-point notation	The f floating-point modifier instructs the functions to convert a value to string format in floating-point notation. This is the default.
t	Truncate	The t floating-point modifier indicates that in floating-point to integer transformations, the functions truncate instead of round the floating-point value. This is the default.
r	Round	The r floating-point modifier indicates that in floating-point to integer transformations, the functions round instead of truncate the floating-point value. The default method is truncation.

Table 2-5. Formatting String Modifiers (%s)

Modifier	Meaning	Description
i*n*	Specify array offset	The i string modifier specifies an offset within a string. It indicates the location within the string where processing begins. *n* is the zero-based index of the first byte to process. Thus, %s[i2] applied to a target string begins placing data in the third byte of the string.
a	Append	When applied to a target format specifier, the a string modifier specifies that all formatted data be appended to the target string, beginning at the first occurrence of an ASCII NUL in the target string.
w*n*	Specify string size	When applied to a source format specifier, the w string modifier specifies the maximum number of bytes to consume from the string argument. You can enter any non-negative value here. The default is zero, which indicates that the entire string should be consumed. When modifying a target format specifier, the w string modifier specifies the exact number of bytes to store in the string, excluding the terminating ASCII NUL. If *n* is zero or omitted, the functions store as many bytes as the sources call for. When *n* is greater than the number of bytes available from the source, the remaining bytes are filled with ASCII NULs if you use the q modifier or blanks if you do not use the q modifier. When you use the w string modifier in conjunction with the a string modifier, *n* indicates the number of bytes to append to the string excluding the terminating ASCII NUL.
q	Append NULs	When applied to a target string in conjunction with the w string modifier, the q string modifier specifies that unfilled bytes at the end of the target string be set to ASCII NULs instead of blanks.

Table 2-5. Formatting String Modifiers (%s) (Continued)

Modifier	Meaning	Description
`t`*n*	Terminate on character	When applied to a source string, the `t` string modifier specifies that the source string terminates on the first occurrence of the character `n`, where *n* is the ASCII value of the character. Thus, `%s[t44]` stops the reading of the source string on an ASCII comma. Using `%s[t44]` and the source string `Hello, World!` as an example, the functions place `Hello` into the target. More than one `t` modifier can occur in the same specifier, in which case the string terminates when any of the terminators occur. If no `t` modifier exists, reading of the source string stops on an ASCII NUL. This modifier has no effect when you apply it to the target specifier.
`t-`	Terminate when full	This is similar to `t`*n* except that it specifies that there are *no* terminating characters. Reading of the source string terminates when the target is full or when the functions have read the number of bytes specified with the `w` modifier.
`t#`	Terminate on number	This is equivalent to repeating the `t` modifier with the ASCII values of the characters `+`, `-`, and `0`–`9`. It instructs the functions to stop the reading of the source string on occurrence of a numeric expression. If you use `%s[t#]` with the source string `ab567`, the functions place `ab` in the target.

Fmt, FmtFile, FmtOut—Asterisks (*) Instead of Constants in Format Specifiers

Often, a format specifier requires one or more integer values. The format specifier for an integer array, for example, requires the number of elements (`rep`). You can use constants for these integer values in format specifiers. Also, you can specify an integer value using an argument in the argument list. When you use this method, substitute an asterisk (`*`) for the constant in the format specifier.

Use the asterisk in the following format specifier elements:

`rep`	For integer or floating-point arrays
`i`*n*	For integer or floating-point arrays, or strings
`w`*n*	For any format specifier
`p`*n*	For floating-point specifiers only
`e`*n*	For floating-point specifiers only
`r`*n*	For integer specifiers only

When you use one or more asterisks instead of constants in a *target* specifier, the arguments that correspond to the asterisks must appear *after* the format string in the same order as the corresponding asterisks appear in the format specifier.

When you use one or more asterisks instead of constants in a *source* specifier, the arguments that correspond to the asterisks must *precede* the source argument and must be in the same order as the corresponding asterisks in the format specifier.

Fmt, FmtFile, FmtOut—Literals in the Format String

Literal characters that appear in a formatting function format string indicate that the literal characters are to be combined with the source parameters in the appropriate positions. They do not correspond to any source parameters but are copied directly into the target item.

Because the left side of the `<` symbol must be a single format specifier, literal characters must be on the right side of the symbol. Literals on the left side or more than one format specifier on the left side result in a `-1` error, indicating a faulty format string. You then can use `GetFmtErrNdx` to determine exactly where the error lies in the format string.

The characters `%`, [,], `<`, and `>` have special meaning in the format strings. To specify that these characters be taken literally, precede them with `%`.

Scanning Functions

The following information is a brief description of the three scanning functions.

- `n = Scan (source, formatstring, targetptr1, ..., targetptrn);`

  `Scan` inspects the `source` argument and applies transformations to it according to descriptions in the `formatstring` argument. `Scan` places the results of the transformations into the `targetptr1 ... targetptrn` arguments.

- `n = ScanFile (handle, formatstring, targetptr1, ..., targetptrn);`

  `ScanFile` reads data from the file that corresponds to the `handle` argument and applies transformations to it according to descriptions in the `formatstring` argument. `ScanFile` places the results of the transformations into the `targetptr1 ... targetptrn` arguments.

- `n = ScanIn (formatstring, targetptr1, ..., targetptrn);`

  `ScanIn` reads data from the Standard Input and applies transformations to it according to descriptions in the `formatstring` argument. `ScanIn` places the results of the transformations into the `targetptr1 ... targetptrn` arguments.

Each of these scanning functions returns the number of `target` format specifiers satisfied. If an error exists in the format string, the functions return `-1`.

The scanning functions break apart a source item into component parts and store the parts into parameters passed to the function. The only difference among the three functions is the location of the source data. For `Scan`, the source item is a data item in memory that you pass to the function. For `ScanFile`, the source item is a file, whose handle you pass as the first argument. Call `OpenFile` to obtain a file handle. For `ScanIn`, the function takes the source from the Standard Input, typically the keyboard, and omits the source argument present in the other two functions.

You must pass all target parameters for `Scan` by reference.

Scanning Functions—Format String

Consider the following scanning function:

```
n = Scan(source, formatstring, targetptr1, ..., targetptrn);
```

where `formatstring` contains the information to transform the `source` argument to the `targetptr` arguments.

Format strings for the scanning functions follow the form:

```
"source_spec > target_specs_and_literals"
```

where `source_spec` is a format specifier that describes the nature of the source parameter and `target_specs_and_literals` is a sequence of format specifiers and literal characters that indicate how to divide and reformat the source argument into the desired target.

Examples of format strings for the scanning functions are as follows:

```
"%s > %i"
"%s > %20f[w10x]"
```

The character '>' serves as a visual reminder of the direction of the data transformation and separates the single source format specifier from the target format specifiers and literals. You can omit the source format specifier, in which case the functions assume a `%s` string format. If you omit the source format specifier, you can omit the '>' character or retain it for clarity.

Notice that the source format specifier is located to the left of the '>' symbol, just as the source parameter is located to the left of the format string. Likewise, the target format specifiers are located to the right of the '>' symbol, just as the target parameters are located to the right of the format string.

Format specifiers describe the inputs and outputs of data transformations. Each format specifier is of the following form:

```
%   [ rep ]   formatcode   [[ modifiers ]]
```

The character `%` introduces all format specifiers. `rep` indicates how many times the format repeats with respect to the arguments. `formatcode` is a code character that indicates the nature of the data items the functions format. `modifiers` is an optional bracket-enclosed sequence of codes that further describe the data format.

Examples of format specifiers are a follows:

```
%s[t59]   %100i[z]   %f
```

☞ **Note** `rep` ***is not allowed when*** `formatcode` ***is*** `s` ***or*** `l` ***(string).***

`formatcode` is specified with one of the following codes in Table 2-6:

Table 2-6. Codes That Specify formatcode

formatcode	Meaning	Description
`s`	String	As a source or target specifier, this indicates that the corresponding parameter is a character string. As a source specifier, the number of bytes of the source parameter that the functions consume depends on the target specifier. If the target specifier is `%s`, the functions consume bytes until they encounter a termination character. Refer to the `t` modifier for strings for more information on termination characters. If the target specifier is one of the numeric formats, the functions consume bytes as long as the bytes correspond to the pattern for the particular numeric item the functions are converting. The functions skip leading spaces and tabs unless you use the `y` modifier. **Note**: When the functions fill in a target string, they always place an ASCII NUL in the string after the last byte.
`l`	String	This is allowed only as a source specifier. It is the same as the `%s` specifier, except that the function consumes bytes from the source argument only until it encounters a linefeed. You can modify with `c`, as in `%l[c]`, to tell the functions to use a comma as the target string terminator in place of white space characters.

Table 2-6. Codes That Specify formatcode (Continued)

formatcode	Meaning	Description
`i`	Integer	This source or target specifier indicates that the corresponding parameter is an integer or, if `rep` is present, an integer array. As a source specifier in conversions to string formats, the functions convert the integer into digits of the specified radix. The default is decimal. As a target specifier in conversions from string format, the functions consume bytes of the source parameter as long as they match the pattern of integer ASCII numbers in the appropriate radix or until the functions encounter the end of the string. The functions convert the scanned characters to integer values and place them into the corresponding target parameter, which is an integer array or integer you pass by reference. The pattern for integer ASCII numbers consists of an optional sign (`+` or `-`), followed by a series of one or more digits in the appropriate radix. The decimal digits are `01234 56789`. The octal digits are `01234567`. The hexadecimal digits are `0123456789ABCDEFabcdef`.
`x`	Integer (hexadecimal)	This specifier indicates a `%i` format with hexadecimal radix.
`o`	Integer (octal)	This specifier indicates a `%i` format with octal radix.
`d`	Integer (decimal)	This specifier is identical to `%i` and is included for compatibility with the C `scanf` family of functions.

Table 2-6. Codes That Specify formatcode (Continued)

formatcode	Meaning	Description
`f`	Real number	As a source or target specifier, this indicates that the corresponding parameter is a real number, or if `rep` is present, a real array. As a source specifier in conversions to string formats, the functions convert the floating-point value into ASCII form. As a target specifier in conversions from string format, the function consumes bytes of the source parameter as long as they match the pattern of floating-point ASCII numbers or until the functions encounter the end of the string. The functions convert the scanned characters to a floating-point value and place them into the corresponding floating-point or floating-point array target parameter. The pattern for floating-point ASCII numbers is an optional sign (`+` or `-`), a series of one or more decimal digits that can contain a decimal point, and an optional exponent that consists of an `E` or `e` followed by an optionally signed decimal integer value.
`c`	Character	As a source specifier, this indicates that the source parameter is an integer with one significant byte or, if `rep` is present, an array of 1-byte integers. As a target specifier, this indicates that the functions consume a byte of the source parameter and place the scanned character directly into the corresponding target parameter, which is an integer you pass by reference.

Scanning Modifiers

`modifiers` are optional codes you use to describe the nature of the source or target data. If you use them, you must enclose the modifiers in square brackets and place them immediately after the format code they modify. If one format specifier requires more than one modifier, enclose all modifiers in the same set of brackets.

A different set of modifiers exists for each possible format specifier as shown in Table 2-7, Table 2-8, and Table 2-9.

Table 2-7. Scanning Integer Modifiers (%i, %d, %x, %o, %c)

Modifier	Meaning	Description
b*n*	Specify length	The b integer modifier specifies in bytes the length of the integer argument or the length of an individual integer array element. The default length is 4 bytes; therefore, simple 4-byte integers do not need this modifier. The modifier `b2` represents short integers. The modifier `b1` represents single-byte integers.
i*n*	Specify array offset	Use the `i` integer modifier to specify an offset within an integer array argument. It indicates the location within the array where processing begins. *n* is the zero-based index of the first element to process. Thus, `%10d[i2]` applied to a source integer array processes the 10 integer values from the third–12th elements of the array. The `i` modifier is valid only if `rep` is present. If you use the `i` modifier with the `z` modifier, *n* is in terms of bytes.
z	Treat string as integer array	The `z` integer modifier indicates that the data type of the corresponding argument is a string. Nevertheless, the functions treat the data in the string as an integer array. The `z` modifier is valid only if `rep` is present.
r*n*	Specify radix	The `r` integer modifier specifies the radix of the integer argument, which is important if the functions convert the integer from a string format. Legal radixes are 8 (octal), 10 (decimal, the default), 16 (hexadecimal), and 256 (a special radix that represents single 8-bit ASCII characters).
w*n*	Specify string size	The `w` integer modifier specifies the exact number of bytes occupied by a string representation of the integer argument, in the event that the functions convert the integer from a string format. You can enter any non-negative value here. If `n` is less than the number of digits required to represent the integer, the functions insert an asterisk (`*`) into the string to signify an overflow. The default for *n* is zero, which indicates that the integer can occupy as much room as necessary.

Table 2-7. Scanning Integer Modifiers (%i, %d, %x, %o, %c) (Continued)

Modifier	Meaning	Description
s	Specify as two's complement	The s integer modifier indicates that the functions consider the integer argument a signed two's complement number. This is the default interpretation of integers, so the functions do not require the s modifier.
u	Specify as non-negative	The u integer modifier indicates that the functions consider the integer to be a non-negative integer.
x	Discard terminator	The x integer causes the functions to discard the character that terminated the numeric data. In this way, the functions can skip the terminator characters when reading lists of numeric input. Thus, %3i[x] reads three integer numbers, disregarding the terminator character that appears after each one. You can use this specifier to scan the string 3, 7, -32.
d	Discard data	When applied to a target specifier, the d integer modifier indicates that no target argument exists to correspond to the target specifier. The data the functions would place in the target argument is discarded instead. The count the functions return *includes* the target specifier even if you use the d modifier.
o*nnnn*	Specify byte ordering	You use the o integer modifier to describe the byte ordering of raw data so that LabWindows/CVI can map it to the byte order appropriate for the Intel (PC) or Motorola (SPARCstation) architecture. The number of *n*'s must be equal to the byte size of the integer argument as specified by the b*n* modifier, which must precede the o modifier. In the case of a four-byte integer, o0123 indicates that the bytes are in ascending order of precedence (Intel style), and o3210 indicates that the bytes are in descending order of precedence (Motorola style).

When using the Scan function to transfer raw instrument data to or from a C integer or integer array, you must use the o modifier on the buffer that contains the raw instrument data. Do not use the o modifier on the buffer that contains the C integer or integer array. LabWindows/CVI determines the byte ordering of the buffer without the o modifier based on the architecture on which your program is running.

For example, if your GPIB instrument sends 2-byte binary data in Intel-byte order, your code should appear as follows:

```
short int instr_buf[100];
short int prog_buf[100];
status = ibrd (ud, instr_buf, 200);
Scan (instr_buf, "%100d[b2o01]>%100d", prog_buf);
```

If, instead, your GPIB instrument sends 2-byte binary data in Motorola-byte order, `Scan` should appear as follows:

```
Scan (instr_buf, "%100d[b2o10]>%100d", prog_buf);
```

In either case, you use the `o` modifier only on the buffer that contains the raw data from the instrument (`instr_buf`). LabWindows/CVI ensures that the program buffer (`prog_buf`) is in the proper byte order for the host processor.

Note ***When you use both the*** `bn` ***and*** `on` ***modifiers on an integer specifier, the*** `bn` ***modifier must be first.***

Table 2-8. Scanning Floating-Point Modifiers (%f)

Modifier	Meaning	Description
`bn`	Specify length	The `b` floating-point modifier specifies the length of the floating-point argument, or the length of an individual array element, in bytes. The default length is 8 bytes; therefore, double-precision values do not need this modifier. Single-precision floating-point values are indicated by `b4`. The only valid values for *n* are `8` and `4`.
`in`	Specify array offset	The `i` floating-point modifier specifies an offset within a floating-point array argument. It indicates the location within the array where processing begins. *n* is the zero-based index of the first element to process. Thus, `%10f[i2]` applied to a source floating-point array processes the 10 floating-point values from the third–12th elements of the array. The `i` modifier is valid only if `rep` is present. If you use the `i` modifier with the `z` modifier, *n* is in terms of bytes.

Table 2-8. Scanning Floating-Point Modifiers (%f) (Continued)

Modifier	Meaning	Description
`z`	Treat string as floating point	The `z` floating-point modifier indicates that the data type of the corresponding argument is a string. Nevertheless, the functions treat the data in the string as a floating-point array. The `z` modifier is valid only if `rep` is present.
`w`*n*	Specify string size	The `w` floating-point modifier specifies the exact number of bytes occupied by a string representation of the floating-point argument, in the event that the functions convert the value from a string format. You can enter any non-negative value here. If *n* is less than the number of digits required to represent the floating-point number, the functions insert an asterisk (`*`) into the string to signify an overflow. The default for *n* is zero, which indicates that the value can occupy whatever space is necessary.
`p`*n*	Specify precision	The `p` floating-point modifier specifies the number of digits to the right of the decimal point in a string representation of the floating-point number. You can lose significant digits by attempting to conform to the precision specification. If you omit the `p`*n* modifier, the default is `p6`. The `p` modifier is valid for sources only.
`e`*n*	Specify as scientific notation	The `e` floating-point modifier indicates that the string representation of the floating-point value is in scientific notation. If you omit the modifier, the functions use non-scientific notation. *n* is optional and specifies the number of digits to use in the exponent. For example, `%f[e2]` causes the functions to format 10.0 as 1.0e+01. If you omit *n*, the functions use a default of three. The `e` modifier is valid for sources only.
`f`	Specify as floating point	The `f` floating-point modifier indicates that the string representation of the floating-point value is not in scientific notation. This is the default even when the `f` modifier is absent.

Table 2-8. Scanning Floating-Point Modifiers (%f) (Continued)

Modifier	Meaning	Description
x	Discard terminator	The x floating-point modifier causes the functions to discard the character that terminated the numeric data. In this way, the functions can skip terminator characters when reading lists of numeric input. Thus, `%3f[x]` reads three floating-point numbers, disregarding the terminator character that appears after each one. You can use this specifier to scan the string `3.5, 7.6, -32.4`.
d	Discard data	When applied to a target specifier, the d modifier indicates no target argument exists to correspond to the target specifier. The data the functions would place in the target argument is discarded instead. The count the functions return *includes* the target specifier even if you use the d modifier.

Table 2-9. Scanning String Modifiers (%s)

Modifier	Meaning	Description
i*n*	Specify array offset	The i string modifier specifies an offset within a string. It indicates the location within the string where processing begins. *n* is the zero-based index of the first byte to process. Thus, `%s[i2]` applied to a target string begins placing data in the third byte of the string.
a	Append	When applied to a target format specifier, the a string modifier specifies that all formatted data be *appended* to the target string, beginning at the first occurrence of an ASCII NUL in the target string.

Table 2-9. Scanning String Modifiers (%s) (Continued)

Modifier	Meaning	Description
w*n*	Specify string size	When applied to a source format specifier, the w string modifier specifies the maximum number of bytes from the source string to use for filling the target arguments. You can enter any non-negative value here. The default is zero, which indicates that the entire string can be used. In this case, the ScanFile and ScanIn functions consume the entire source string even if the w modifier restricts the number of bytes used to fill in the target arguments. When modifying a target format specifier, the w modifier specifies the exact number of bytes to store in the string, excluding the terminating ASCII NUL. If *n* is zero or omitted, the functions store as many bytes as are called for by the sources. When *n* is greater than the number of bytes available from the source, the remaining bytes are filled with ASCII NULs if you use the q modifier or blanks if you do not use the q modifier. When you use the w modifier in conjunction with the a modifier, *n* indicates the number of bytes to append to the string excluding the terminating ASCII NUL.
q	Append NULs	When applied to a target string in conjunction with the w string modifier, the q string modifier specifies that the functions set unfilled bytes at the end of the target string to ASCII NULs instead of blanks.
y	Append with spacing	When the source is a string and you apply the y modifier to a target string format specifier, the functions fill the target string with bytes from the source string without skipping leading spaces or tabs.

Table 2-9. Scanning String Modifiers (%s) (Continued)

Modifier	Meaning	Description
`t`*n*	Terminate on character	When applied to a source string, the `t` modifier specifies that the source string terminates on the first occurrence of the character *n*, where *n* is the ASCII value of the character. Thus, `%s[t44]` stops the reading of the source string on an ASCII comma. More than one `t` modifier can occur in the same specifier, in which case the string terminates when any of the terminators occur. If no `t` modifier exists, the functions stop reading the source string on an ASCII NUL. When applied to a target string that the functions fill from a source string, the `t` modifier specifies that the functions stop filling the target on the first occurrence of the character *n*, where *n* is the ASCII value of the character. Thus, `%s[t59]` causes the functions to stop reading the source string on an ASCII semicolon. More than one `t` modifier can occur in the same specifier, in which case the functions stop filling the target when any of the terminators occur. If no `t` modifier exists, the functions stop filling the target on any whitespace character.
`t-`	Terminate when full	This is similar to `t`*n* except that it specifies that there are *no* terminating characters. When applied to a source string, `t-` specifies that the functions stop reading the source string when all the targets are full or when the functions have read the number of bytes you specify with the `w` modifier. When applied to a target string, `t-` specifies that the functions stop filling the target string when the source is exhausted or when the functions have placed into the target the number of bytes you specify with the `w` modifier.
`t#`	Terminate on number	This is equivalent to repeating the `t` modifier with the ASCII values of the characters `+`, `-`, and `0`–`9`. When applied to a source (target), it specifies that the functions stop reading the source string, or filling the target string, upon occurrence of a numeric expression. If you use `%s>%s[t#]%d` with the source string `"ab567"`, the functions place `"ab"` in the first target and the integer `567` in the second target.

Table 2-9. Scanning String Modifiers (%s) (Continued)

Modifier	Meaning	Description
x	Discard terminator	When applied to a target string, the x modifier specifies that the functions discard the terminating character before the next target is filled in. Using %s>%s[xt59]%s[xt59] with the source string "abc;XYZ;", the functions place "abc" in the first target and "XYZ" in the second target.
d	Discard data	When applied to a target specifier, the d modifier indicates that no target argument corresponds to the target specifier. The data that the functions otherwise place in the target argument is discarded instead. The count the functions return *includes* the target specifier even if you use the d modifier.

Scan, ScanFile, ScanIn—Asterisks (*) Instead of Constants in Format Specifiers

Often, a format specifier requires one or more integer values. The format specifier for an integer array, for example, requires the number of elements (rep). You can use constants for these integer values in format specifiers. You can also specify an integer value using an argument in the argument list. When you use this method, substitute an asterisk (*) for the constant in the format specifier

Use the asterisk in the following format specifier elements:

rep For integer or floating-point arrays

i*n* For integer or floating-point arrays, or strings

w*n* For any format specifier

p*n* For floating-point specifiers only

e*n* For floating-point specifiers only

r*n* For integer specifiers only

When you use one or more asterisks instead of constants in a *source* specifier, the arguments that correspond to the asterisks must appear *after* the format string in the same order as the corresponding asterisks appear in the format specifier.

When you use one or more asterisks instead of constants in a *target* specifier, the arguments that correspond to the asterisks must *precede* the target argument and must be in the same order as the corresponding asterisks in the format specifier.

Scan, ScanFile, ScanIn—Literals in the Format String

Literal characters that appear in a scanning function format string indicate that the functions expect the literal characters in the source parameter. The functions do not store them in any target parameter but skip over them when encountered. If a literal character you specify in the format string fails to appear in the source in the expected position, the scanning function returns without processing the target specifiers that appear to the right of the unmatched literal. The scanning functions return the number of target parameters the input actually fulfilled. `NumFmtdBytes` returns the number of bytes consumed from the source parameter.

Because the left side of the > symbol must be a single format specifier, literal characters, if present, must be on the right side of the symbol. Literals on the left side, or more than one format specifier on the left side, result in a `-1` error, indicating a faulty format string. If you receive this error, you can use `GetFmtErrNdx` to determine exactly where in the format string the error lies.

The characters %, [,], <, and > have special meaning in the format strings. To specify that these characters be taken literally, precede them with %.

Formatting and I/O Library Programming Examples

This section contains examples of program code that use the Formatting and I/O Library functions. The formatting and scanning functions are the basis of most of the examples.

The `Fmt`/`FmtFile`/`FmtOut` examples are logically organized as shown:

Integer to string
Short integer to string
Real to string in floating-point notation
Real to string in scientific notation
Integer and real to string with literals
Two integers to ASCII file with error checking
Real array to ASCII file in columns and with comma separators
Integer array to binary file, assuming a fixed number of elements
Real array to binary file, assuming a fixed number of elements
Real array to binary file, assuming a variable number of elements
Variable portion of a real array to a binary file
Concatenating two strings
Appending to a string
Creating an array of filenames
Writing a line that contains an integer with literals to the standard output
Writing to the standard output without a linefeed or carriage return

The `Scan`/`ScanFile`/`ScanIn` examples are logically organized as shown:

String to integer
String to short integer
String to real
String to integer and real
String to string
String to integer and string
String to real, skipping over non-numeric characters in the string
String to real, after finding a semicolon in the string
String to real, after finding a substring in the string
String with comma-separated ASCII numbers to real array
Scanning strings that are not null-terminated
Integer array to real array
Integer array to real array with byte swapping
Integer array that contains 1-byte integers to real array
Strings that contain binary integers to integer array
Strings that contain an IEEE-format real number to a real variable
ASCII file to two integers with error checking
ASCII file with comma-separated numbers to real array, with the number of elements at the beginning of file
Binary file to integer array, assuming a fixed number of elements
Binary file to real array, assuming a fixed number of elements
Binary file to real array, assuming a variable number of elements
Reading an integer from the standard input
Reading a string from the standard input
Reading a line from the standard input

Fmt/FmtFile/FmtOut Examples in C

This section contains examples of program code that use `Fmt`, `FmtFile`, and `FmtOut` from the Formatting and I/O Library. To eliminate redundancy, error checking on I/O operations has been omitted from all the examples in this section except the *Two Integers to ASCII File with Error Checking* example.

Integer to String

```
char buf[10];
int a;
a = 16;
Fmt (buf, "%s<%i", a); /* result: "16" */
a = 16;
Fmt (buf, "%s<%x", a); /* result: "10" */
a = 16;
```

```
Fmt (buf, "%s<%o", a); /* result: "20" */
a = -1;
Fmt (buf, "%s<%i", a); /* result: "-1" */
a = -1;
Fmt (buf, "%s<%i[u]", a); /* result: "4294967295" */
a = 1234;
Fmt (buf, "%s<%i[w6]", a); /* result: "  1234" */
a = 1234;
Fmt (buf, "%s<%i[w6p0]", a); /* result: "001234" */
a = 1234;
Fmt (buf, "%s<%i[w2]", a); /* result: "*4" */
```

The results shown are the contents of `buf` after each call to `Fmt`. The last call demonstrates what occurs when the `w` modifier specifies a width that is too small.

Short Integer to String

```
char buf[20];
short a;
a = 12345;
Fmt (buf, "%s<%i[b2]", a); /* result: "12345" */
a = -1;
Fmt (buf, "%s<%i[b2]", a); /* result: "-1" */
a = -1;
Fmt (buf, "%s<%i[b2u]", a); /* result: "65535" */
a = 12345;
Fmt (buf, "%s<%i[b2w7]", a); /* result: "  12345" */
a = 12345;
Fmt (buf, "%s<%i[b2w7p0]", a); /* result: "0012345" */
a = 12345;
Fmt (buf, "%s<%i[b2w4]", a); /* result: "*345" */
```

The results shown are the contents of `buf` after each call to `Fmt`. The last call demonstrates what occurs when the `w` modifier specifies a width that is too small.

Real to String in Floating-Point Notation

```
char buf[30];
double x;
x = 12.3456789;
Fmt (buf, "%s<%f", x); /* result: "12.345679" */
x = 12.3456789;
Fmt (buf, "%s<%f[p2]", x); /* result: "12.35" */
x = 12.3456789;
Fmt (buf, "%s<%f[p10]", x); /* result: "12.3456789000" */
x = 12.345;
Fmt (buf, "%s<%f", x); /* result: "12.345" */
x = 12.345;
Fmt (buf, "%s<%f[p0]", x); /* result: "12" */
x = 12.345;
Fmt (buf, "%s<%f[p6]", x); /* result: "12.345000" */
x = -12.345;
Fmt (buf, "%s<%f[w12]", x); /* result: "     -12.345" */
x = -12.3456789;
Fmt (buf, "%s<%f[w6]", x); /* result: "-12.3*" */
x = 0.00000012;
Fmt (buf, "%s<%f[p8]", x); /* result: "0.00000012" */
x = 0.00000012;
Fmt (buf, "%s<%f", x); /* result: "1.2e-007" */
x = 4.5e050;
Fmt (buf, "%s<%f", x); /* result: "4.5e050" */
```

The results shown are the contents of `buf` after each call to `Fmt`. The last two calls demonstrate that `Fmt` sometimes forces very large and very small values into scientific notation even when the `e` modifier is absent.

Real to String in Scientific Notation

```
char buf[20];
double x;
x = 12.3456789;
Fmt (buf, "%s<%f[e]", x); /* result: "1.234568e+001" */
x = 12.3456789;
Fmt (buf, "%s<%f[ep2]", x); /* result: "1.23e+001" */
x = 12.3456789;
Fmt (buf, "%s<%f[e2p2]", x); /* result: "1.23e+01" */
x = 12.345;
Fmt (buf, "%s<%f[e]", x); /* result: "1.2345e+001" */
x = 12.345;
Fmt (buf, "%s<%f[ep2w12]", x); /* result: "   1.23e+001" */
```

```
x = 12.345;
Fmt (buf, "%s<%f[ep2w6]", x); /* result: "1.23e*" */
```

The results shown are the contents of `buf` after each call to `Fmt`. The last call demonstrates what occurs when the `w` modifier specifies a width that is too small.

Integer and Real to String with Literals

```
char buf[20];
int f, r;
double v;
f = 4;
r = 3;
v = 1.2;
Fmt (buf, "%s<F%iR%i; V%f;", f, r, v);
```

After the `Fmt` call, `buf` contains `"F4R3; V1.2;"`.

Two Integers to ASCII File with Error Checking

```
int a, b, n, file_handle;
a = 12;
b = 456;
file_handle = OpenFile ("FILE.DAT", 2, 0, 1);
if (file_handle < 0) {
   FmtOut ("Error opening file\n");
   exit (1);
}
n = FmtFile (file_handle, "%s<%i %i", a, b);
if (n != 2) {
   FmtOut ("Error writing file\n");
   exit (1);
}
CloseFile (file_handle);
```

`OpenFile` opens the file `FILE.DAT` as an ASCII file for writing only. If `OpenFile` succeeds, it returns a file handle with a positive integer value. `FmtFile` writes the ASCII representation of two integer values to the file. If `FmtFile` succeeds, it returns `2` because the format string contains two source specifiers.

Real Array to ASCII File in Columns and with Comma Separators

```
double x[100];
int file_handle, i;
file_handle = OpenFile ("FILE.DAT", 2, 0, 1);
```

```
for (i=0; i < 100; i++) {
    FmtFile (file_handle, "%s<%f[w15],", x[i]);
    if ((i % 5) == 4)
        WriteFile (file_handle, "\n", 1);
}
CloseFile (file_handle);
```

The `FmtFile` call writes the ASCII representation of a real array element to the file, followed by a comma. The `w` modifier specifies that the number be right-justified in a 15-character field. The `WriteFile` call writes a linefeed to the file after every fifth call to `FmtFile`. Because the file is opened in ASCII mode, `FmtFile` automatically writes the linefeed as a linefeed/carriage return combination.

Note ***If the format string is*** `"%s[w15]<%f,"`**,** `FmtFile` ***left-justifies the number and the comma together in a 15-character field.***

Integer Array to Binary File, Assuming a Fixed Number of Elements

```
int readings[100];
int file_handle, nbytes;
file_handle = OpenFile ("FILE.DAT", 2, 0, 0);
FmtFile (file_handle, "%100i<%100i", readings);
nbytes = NumFmtdBytes ();
CloseFile (file_handle);
```

The `FmtFile` call writes all 100 elements of the `readings` integer array to a file in binary form. If the `FmtFile` call succeeds, `nbytes` = 200 (100 integers, 2 bytes per integer).

Real Array to Binary File, Assuming a Fixed Number of Elements

```
double waveform[100];
int file_handle, nbytes;
file_handle = OpenFile ("FILE.DAT", 2, 0, 0);
FmtFile (file_handle, "%100f<%100f", waveform);
nbytes = NumFmtdBytes ();
CloseFile (file_handle);
```

The `FmtFile` call writes all 100 elements of the `waveform` real array to a file in binary form. If the `FmtFile` call succeeds, `nbytes` = 800 (100 integers, 8 bytes per real number).

Real Array to Binary File, Assuming a Variable Number of Elements

```
void StoreArray (double x[], int count, char filename[])
{
   int file_handle;
   file_handle = OpenFile (filename, 2, 0, 0);
   FmtFile (file_handle, "%*f<%*f", count, count, x);
   CloseFile (file_handle);
}
```

This example shows how you can use a function to write an array of real numbers to a binary file. The function parameters are a real array, the number of elements to be written, and the filename.

The `FmtFile` call writes the first `count` elements of `x` to a file in binary form. `FmtFile` matches the two asterisks (`*`) in the format string to `count`. For instance, if `count` is 100, the format string is equivalent to `%100f<%100f`.

Variable Portion of a Real Array to a Binary File

```
void StoreSubArray (double x[], int start, int count, char filename[])
{
   int file_handle;
   file_handle = OpenFile (filename, 2, 0, 0);
   FmtFile (file_handle, "%*f<%*f[i*]", count, count, start, x);
   CloseFile (file_handle);
}
```

This example is an extension of the previous example. The function writes a variable number of elements of a real array to a file. Instead of beginning at the first element of the array, you pass a starting index to the function.

The `FmtFile` call writes `count` elements of `x`, starting from `x[start]`, to a file in binary form. `FmtFile` matches the first two asterisks (`*`) in the format string to `count`. `FmtFile` matches the third asterisk to `start`. For instance, if `count` is 100 and `start` is 30, the format string is equivalent to `%100f<%100f[i30]`. Because the `i` modifier specifies a zero-based index into the real array, `FmtFile` writes the array elements from `x[30]` through `x[129]` to the file.

Concatenating Two Strings

```
char buf[30];
int wave_type, signal_output;
char *wave_str, *signal_str;
int nbytes;
wave_type = 1;
signal_output = 0;
switch (wave_type) {
   case 0:
      wave_str = "SINE;"
      break;
   case 1:
      wave_str = "SQUARE;"
      break;
   case 2:
      wave_str = "TRIANGLE;"
      break;
}
switch (signal_output) {
   case 0:
      signal_str = "OUTPUT OFF;"
      break;
   case 1:
      signal_str = "OUTPUT ON;"
      break;
}
Fmt (buf, "%s<%s%s", wave_str, signal_str);
nbytes = NumFmtdBytes ();
```

The two `switch` constructs assign constant strings to the string variables `wave_str` and `signal_str`. The `Fmt` call concatenates the contents of `wave_str` and `signal_str` into `buf`. After the call, `buf` contains `"SQUARE;OUTPUT OFF;"`. `NumFmtdBytes` returns the number of bytes in the concatenated string.

Appending to a String

```
char buf[30];
int wave_type, signal_output;
int nbytes;
switch (wave_type) {
   case 0:
      Fmt (buf, "%s<SINE;");
      break;
   case 1:
      Fmt (buf, "%s<SQUARE;");
      break;
   case 2:
      Fmt (buf, "%s<TRIANGLE;");
      break;
}
switch (signal_output) {
   case 0:
      Fmt (buf, "%s[a]<OUTPUT OFF;");
      break;
   case 1:
      Fmt (buf, "%s[a]<OUTPUT ON;");
      break;
}
nbytes = StringLength (buf);
```

This example shows how to append characters to a string without writing over the existing contents of the string. The first `switch` construct writes one of three strings into `buf`. The second `switch` construct appends one of two strings to the string already in `buf`. After the call, `buf` contains `"SQUARE;OUTPUT OFF;"`. Notice that the `a` modifier applies to the target specifier.

`StringLength` returns the number of bytes in the resulting string. In this case, `Fmt` uses `StringLength` instead of `NumFmtdBytes`, because `NumFmtdBytes` returns only the number of bytes appended.

Creating an Array of Filenames

```
char *fname_array[4];
int i;
for (i=0; i < 4; i++){
   frame_array[i] = malloc(14);
   Fmt (fname_array[i], "%s<FILE%i[w4p0].DAT", i);
}
```

To allocate the space for each filename in the array, you must assign a separate constant string to each array element. Use `Fmt` to format each filename. The resulting filenames are `FILE0000.DAT`, `FILE0001.DAT`, `FILE0002.DAT`, and `FILE0003.DAT`.

Writing a Line That Contains an Integer with Literals to the Standard Output

```
int a, b;
a = 12;
b = 34;
FmtOut ("%s<A = %i\n", a);
FmtOut ("%s<B = %i\n", b);
```

In this example, the output is as follows:

```
A = 12
B = 34
```

Writing to the Standard Output without a Linefeed or Carriage Return

```
char *s;
int b;
double c;
a = "One";
FmtOut ("%s<%s", a);
b = 2;
FmtOut ("%s<%i", b);
c = 3.4;
FmtOut ("%s<%f", c);
```

This example demonstrates how to write to the Standard Output without a linefeed or carriage return by omitting the `'\n'` from the format string. The output in this example is as follows:

```
One 2 3.4
```

The following code produces the same output:

```
a = "One";
b = 2;
c = 3.4;
FmtOut ("%s<%s %i %f", a, b, c);
```

Scan/ScanFile/ScanIn Examples in C

This section contains examples of program code that use `Scan`, `ScanFile`, and `ScanIn` from the Formatting and I/O Library. To eliminate redundancy, the examples include no error checking on I/O operations in this section except for the *ASCII File to Two Integers with Error Checking* example.

String to Integer

```
char *s;
int a, n;
s = "32";
n = Scan (s, "%s>%i", &a);          /* result: a = 32, n = 1 */
s = "-32";
n = Scan (s, "%s>%i", &a);          /* result: a = -32, n = 1 */
s = "    +32";
n = Scan (s, "%s>%i", &a);          /* result: a = 32, n = 1 */
s = "x32";
n = Scan (s, "%s>%i", &a);          /* result: a = ??, n = 0 */
```

When locating an integer in a string, `Scan` skips over white space characters such as spaces, tabs, linefeeds, and carriage returns. If `Scan` finds a non-numeric character other than a white space character, `+`, or `-` before the first numeric character, the `Scan` call fails. Thus, `Scan` fails on the `x` in `x32`; it leaves the value unmodified in `a` and returns zero, indicating that no target specifiers were satisfied.

```
s = "032";
n = Scan (s, "%s>%i", &a);          /* result: a = 32, n = 1 */
s = "32a";
n = Scan (s, "%s>%i", &a);          /* result: a = 32, n = 1 */
s = "32";
n = Scan (s, "%s>%o", &a);          /* result: a = 26, n = 1 */
s = "32";
n = Scan (s, "%s>%x", &a);          /* result: a = 50, n = 1 */
```

When you use the `%i` specifier, `Scan` interprets numeric characters as decimal, even when they might appear to be octal (as in `032`) or hexadecimal (as in `32a`). When you use the `%o` specifier, `Scan` always interprets the numeric characters (`01234567`) as octal. When you

use the %x specifier, Scan always interprets the numeric characters (0123456789abcdef) as hexadecimal.

```
s = "32x1";
n = Scan (s, "%s>%i", &a);          /* result: a = 32, n = 1 */
```

Scan considers the occurrence of a non-numeric character (such as the x in 32x1) to mark the end of the integer.

```
s = "32567";
n = Scan (s, "%s>%i[w3]", &a);      /* result: a = 325, n = 1 */
```

The w3 modifier specifies that the function only scans the first 3 bytes of the string.

String to Short Integer

```
char *s;
short a;
int n;
s = "9999";
n = Scan (s, "%s>%i[b2]", &a);      /* result: a = 9999, n = 1 */
s = "23417";
n = Scan (s, "%s>%o[b2]", &a);      /* result: a = 9999, n = 1 */
s = "ffff";
n = Scan (s, "%s>%x[b2]", &a);      /* result: a = 65535, n = 1 */
```

Scan extracts short integers from strings in the same way it extracts integers. The only differences are that you must use the b2 modifier and specify the target argument as a short integer. Refer to the *String to Integer* example earlier in this section for more information on using Scan to extract integers and short integers from strings.

String to Real

```
char *s;
double x;
int n;
s = "12.3";
n = Scan (s, "%s>%f", &x);          /* result: x = 12.3, n = 1 */
s = "-1.23e+1";
n = Scan (s, "%s>%f", &x);          /* result: x = -1.23, n = 1 */
s = "1.23e-1";
n = Scan (s, "%s>%f", &x);          /* result: x = 0.123, n = 1 */
```

When locating a real number in a string, Scan accepts either floating-point notation or scientific notation.

```
s = "    12.3";
n = Scan (s, "%s>%f", &x);          /* result: x = 12.3, n = 1 */
s = "p12.3";
n = Scan (s, "%s>%f", &x);          /* result: x = ????, n = 0 */
```

When locating a real number in a string, Scan skips over white space characters. If Scan finds a non-numeric character other than a white space character, +, or - before the first numeric character, the Scan call fails. Thus, Scan fails on the p in p12.3; it leaves the value in x unmodified and returns zero, indicating that no target specifiers were satisfied.

```
s = "12.3m";
n = Scan (s, "%s>%f", &x);          /* result: x = 12.3, n = 1 */
s = "12.3.4";
n = Scan (s, "%s>%f", &x);          /* result: x = 12.3, n = 1 */
s = "1.23e";
n = Scan (s, "%s>%f", &x);          /* result: x = ????, n = 0 */
```

Scan considers the occurrence of a non-numeric character (such as the m in 12.3m) to mark the end of the real number. A second decimal point also marks the end of the number. However, Scan fails on "1.23e" because the value of the exponent is missing.

```
s = "1.2345";
n = Scan (s, "%s>%f[w4]", &x); /* result: x = 1.23, n = 1 */
```

The w4 modifier specifies that the function scans only the first 4 bytes of the string.

String to Integer and Real

```
char *s;
int a, n;
double x;
s = "32   1.23";
n = Scan (s, "%s>%i%f", &a, &x);
          /* result: a = 32, x = 1.23, n = 2 */
s = "32,  1.23";
n = Scan (s, "%s>%i[x]%f", &a, &x);
          /* result: a = 32, x = 1.23, n = 2 */
s = "32,  1.23";
n = Scan (s, "%s>%i%f", &a, &x);
          /* result: a = 32, x = ????, n = 1 */
```

After each of the first two calls to Scan, a = 32, x = 1.23, and n = 2, indicating that two target specifiers were satisfied. In the second call, Scan uses the x modifier is used to discard the separating comma.

In the third call, a comma separator appears after the integer, but the x modifier is absent. Consequently, Scan fails when attempting to find the real number. x remains unmodified, and n = 1, indicating that only one target specifier was satisfied.

String to String

```
char *s;
char buf[10];
int n;
s = "  abc  ";
n = Scan (s, "%s>%s", buf);        /* result: buf = "abc" */
s = "  abc  ";
n = Scan (s, "%s>%s[y]", buf);     /* result: buf = "  abc" */
```

When extracting a substring from a string, Scan skips leading spaces and tabs unless the y modifier is present.

```
s = "a  b  c;  d";
n = Scan (s, "%s>%s", buf);        /* result: buf = "a" */
s = "a  b  c;  d";
n = Scan (s, "%s>%s[t59]", buf);   /* result: buf = "a  b  c" */
```

When Scan extracts a substring from a string and the t modifier is absent, Scan considers the substring to be terminated by a white space character. To include embedded white space in the target string, use the t modifier to change the target string termination character. In the second call to Scan, [t59] changes the termination character to a semicolon (ASCII 59).

```
s = "  abcdefghijklmnop";
n = Scan (s, "%s>%s[w9]", buf);    /* result: buf = "abcdefghi" */
s = "  abc";
n = Scan (s, "%s>%s[w9]", buf);    /* result: buf = "abc "*/
s = "  abc";
n = Scan (s, "%s>%s[w9q]", buf);   /* result: buf = "abc" */
```

The w modifier can be used to prevent Scan from writing beyond the end of a target string. The width specified does not include the ASCII NUL that Scan places at the end of the target string. Therefore, the width you specify should be at least one less than the width of the target character buffer.

When you use the w modifier and the string extracted is smaller than the width specified, Scan fills the remaining bytes in the target string with blanks. However, if you also use the q modifier, ASCII NULs fill the remaining bytes.

String to Integer and String

```
char *s;
char buf[10];
int a, n;
s = "32abc";
n = Scan (s, "%s>%i%s", &a, buf);
          /* result: a = 32, buf = "abc", n = 2 */
s = "32abc";
n = Scan (s, "%s>%i %s", &a, buf);
          /* result: a = 32, buf = ?????, n = 1 */
```

After the first call to `Scan`, `a` = 32, `buf` = `"abc"`, and `n` = 2. Notice there are no spaces in the format string between the two target specifiers. In the second call, there is a space between `%i` and `%s`. Consequently, `Scan` expects a space to occur in `s` immediately after the integer. Because there is no space in `s`, `Scan` fails at that point. It leaves `buf` unmodified and returns `1`, indicating that only one target specifier is satisfied.

Note ***Do not put spaces between specifiers in*** `Scan`***,*** `ScanFile`***, or*** `ScanIn` ***format strings.***

String to Real, Skipping over Non-Numeric Characters in the String

```
char *s;
double x;
int n;
s = "VOLTS = 1.2";
n = Scan (s, "%s>%s[dt#]%f", &x);  /* result: x = 1.2, n = 2 */
s = "VOLTS = 1.2";
n = Scan (s, "%s[i8]>%f", &x);     /* result: x = 1.2, n = 1 */
s = "VOLTS = 1.2";
n = Scan (s, "%s>VOLTS = %f", &x); /* result: x = 1.2, n = 1 */
```

The three different format strings represent different methods for skipping over non-numeric characters. In the first call, the format string contains two target specifiers. In the first specifier (`%s[dt#]`), the `t#` modifier instructs `Scan` to read bytes from `s` until it encounters a number. The `d` modifier tells `Scan` to discard the bytes because no argument corresponds to the specifier. When the `Scan` call succeeds, it returns `2`, indicating that two target specifiers were satisfied, even though only one target argument exists.

In the second call, the source specifier `%s[i8]` instructs `Scan` to ignore the first 8 bytes of `s`. This method works only if the location of the number within `s` is always the same.

In the third call, the format string contains the non-numeric characters literally. This method works only if the non-numeric characters in `s` are always the same.

String to Real, after Finding a Semicolon in the String

```
char *s;
double x;
int n;
s = "TIME 12:45:00;  7.34";
n = Scan (s, "%s>%s[xdt59]%f", &x); /* result: x = 7.34, n = 2 */
```

Some programmable instruments return strings that contain headers that contain both numeric and non-numeric data and are terminated by a particular character, such as a semicolon. This example shows how you can skip such a header.

The format string contains two target specifiers. In the first specifier (`%s[xdt#]`), the `t#` modifier instructs `Scan` to read bytes from `s` until it encounters a number. The `d` modifier indicates that `Scan` must discard the bytes because no argument corresponds to the specifier. The `x` modifier indicates that the semicolon should also be discarded.

When the `Scan` call succeeds, it returns `2`, indicating that two target specifiers were satisfied, even though only one target argument exists.

String to Real, after Finding a Substring in the String

```
char *s;
double x;
int index, n;
s = "HEADER: R5 D6; DATA 3.71E+2";
index = FindPattern (s, 0, -1, "DATA", 0, 0) + 4;
n = Scan (s, "%s[i*]>%f", index, &x);
   /* result: x = 371.0, n = 1 */
```

This example is similar to the previous one except that the portion of the string to skip is terminated by a substring (`DATA`) rather than by a single character. `FindPattern` finds the index where `DATA` begins in `s`. You add four to the index so that it points to the first byte after `DATA`. You then pass the index to `Scan` and match it with the asterisk (`*`) in the format string.

In this example, `FindPattern` returns `15`, and `index` is `19`. When you match `index` to the asterisk in the format string in the `Scan` call, `Scan` interprets the format string as `%s[i19]>%f`. The `i19` indicates that `Scan` should ignore the first 19 bytes of `s`. `Scan` then

extracts the real number from the remaining string, `3.71E+2`, and assigns it to `x`. `Scan` returns `1`, indicating that one target specifier is satisfied.

String with Comma-Separated ASCII Numbers to Real Array

```
char *s;
int n;
double a[5]; /* 5 8-byte real numbers */
s = "12.3, 45, 6.5, -1.3E-2,  4";
n = Scan (s, "%s>%5f[x]", a);
   /* result: a[0] = 12.3, a[1] = 45.0, a[2] = 6.5, */
   /*         a[3] = -0.013, a[4] = 4.0, n = 1*/
```

The `x` modifier causes `Scan` to discard the comma separators.

`Scan` considers an array target to be satisfied when at least one element of the array is filled in. If the source string in this example were `12.3`, only the first element of `a` would be filled in, the other elements would remain unmodified, and `Scan` would return `1`.

Scanning Strings That Are Not Null-Terminated

```
int bd;
double x;
char s[20];
ibrd (bd, s, 15);
Scan (s, "%s[w*]>%f", ibcnt, &x);
```

All the previous examples assume that `s` is a null-terminated string. However, when reading data from programmable instruments using the GPIB and RS-232 Library functions, the data transferred is not null-terminated. This example uses `ibrd` to read up to 15 bytes from a GPIB instrument. The global variable `ibcnt` contains the actual number of bytes transferred. `Scan` uses the value in `ibcnt` in conjunction with the `w` modifier to specify the width of the source string.

For example, if `ibcnt` is 12, the format string is interpreted as `%s[w12]>%f`, causing `Scan` to use only the first 12 bytes of `s`.

The following example is an alternative method for handling strings that are not null-terminated:

```
int bd;
double x;
char s[20];
ibrd (bd, s, 15);
s[15] = 0;                          /* ASCII NUL is 0 */
Scan (s, "%s>%f", &x);
```

This code shows how to insert an ASCII NUL at the end of the transferred bytes. After the assignment, s is null-terminated.

Integer Array to Real Array

```
int ivals[100];
double dvals[100];
Scan (ivals, "%100i>%100f", dvals);
```

Scan converts each integer in ivals to a real number and writes it into dvals.

Integer Array to Real Array with Byte Swapping

```
int ivals[100];
double dvals[100];
Scan (ivals, "%100i[o10]>%100f", dvals);
```

For each integer in ivals, Scan byte-swaps it, converts it to a real number, and writes it into dvals.

Byte swapping is useful when a programmable instrument sends back 2-byte integers with the high byte first, followed by the low byte. When Scan reads this data into an integer array, the placement of the bytes is such that Scan interprets the high byte as the low byte. The o10 modifier specifies that Scan interprets the bytes in the opposite order.

Integer Array That Contains 1-Byte Integers to Real Array

```
int ivals[50];                    /* 100 1-byte integers */
double dvals[100];                /* 100 8-byte real numbers */
Scan (ivals, "%100i[b1]>%100f", dvals);
Scan (ivals, "%100i[b1u]>%100f", dvals);
```

Sometimes, Scan uses each element in an integer array to store two 1-byte integers. This example shows how to unpack the 1-byte integers and store them in a real array. The b1 indicates that each binary integer is only 1 byte long.

The first call to Scan treats the 1-byte integers as signed values, from –128 to +127. The second call includes a u in the format string, which causes Scan to treat the 1-byte integers as unsigned values, from 0 to 255.

Strings That Contain Binary Integers to Integer Array

```
char s[400];                        /* string containing 100 4-byte
                                       integers */
int ivals[100];                     /* 100 4-byte integers */
Scan (s, "%100i[z]>%100i", ivals);
Scan (s, "%97i[zi6]>%97i", ivals);
```

Sometimes `Scan` reads data from a programmable instrument into a character buffer even though it contains binary data. This example shows how to treat a character buffer as an integer array. The format string in each `Scan` call specifies that the source `s` contains an array of 100 integers. The `z` modifier indicates that the source is actually a character buffer.

In some cases, the integer data might not start at the beginning of the character buffer. For instance, the data in the buffer can begin with an ASCII header. In the second call to `Scan`, the `i6` modifier indicates that `Scan` should ignore the first 6 bytes of `s`.

☞ **Note** ***When you use the*** `i` ***modifier in conjunction with a character buffer, the number that follows the*** `i` ***specifies the number of bytes within the buffer to ignore. This is true even when the*** `z` ***modifier is also present. On the other hand, when you use the*** `i` ***modifier in conjunction with an array variable, the number that follows the*** `i` ***indicates the number of array elements to ignore.***

Strings That Contain an IEEE-Format Real Number to a Real Variable

```
char s[100];
double x;
Scan (s, "%1f[z]>%f", &x);
Scan (s, "%1f[zi5]>%f", &x);
```

This example is similar to the previous example except that `s` contains a single binary real number (in IEEE format) rather than an array of binary integers. The format string in each `Scan` call indicates that `Scan` treats the source `s` as a one-element array of real numbers. The `z` modifier indicates that the source is actually a character buffer. The repetition count of `1` in the format string is required; otherwise, `Scan` does not accept the `z` modifier.

The first call to `Scan` assumes that the real number is at the beginning of `s`. The second call assumes that the real number starts at the sixth byte of `s`. The `i5` modifier causes `Scan` to ignore the first 5 bytes of `s`.

ASCII File to Two Integers with Error Checking

```
int file_handle, n, a, b;
file_handle = OpenFile ("FILE.DAT", 1, 2, 1);
if (file_handle < 0) {
   FmtOut ("Error opening file\n");
   exit (1);
}
n = ScanFile (file_handle, "%s>%i%i", &a, &b);
if (n != 2) {
   FmtOut ("Error reading file\n");
   exit (1);
}
CloseFile (file_handle);
```

`OpenFile` opens the file `FILE.DAT` as an ASCII file for reading only. If `OpenFile` succeeds in opening the file, it returns a file handle with a positive integer value. `ScanFile` reads the ASCII representation of two integer values from the file. If `ScanFile` succeeds, it returns `2`, indicating that two target specifiers were satisfied.

ASCII File with Comma-Separated Numbers to Real Array, with Number of Elements at Beginning of File

```
double values[1000];
int file_handle, count;
file_handle = OpenFile ("FILE.DAT", 1, 2, 1);
ScanFile (file_handle, "%s>%i", &count);
if (count > 1000) {
   FmtOut ("Count too large\n");
   exit(1);
}
ScanFile (file_handle, "%s>%*f[x]", count, values);
CloseFile (file_handle);
```

The first `ScanFile` call reads the number of elements into the integer variable `count`. If the value in `count` exceeds the number of elements in the real array `values`, `ScanFile` reports an error. Otherwise, the second `ScanFile` call matches `count` to the asterisk (`*`) in the format string. It then reads the correct number of elements into `values`. The `x` modifier causes `ScanFile` to discard the comma separators.

Binary File to Integer Array, Assuming a Fixed Number of Elements

```
int readings[100];
int file_handle, nbytes;
file_handle = OpenFile ("FILE.DAT", 1, 2, 0);
ScanFile (file_handle, "%100i>%100i", readings);
nbytes = NumFmtdBytes ();
CloseFile (file_handle);
```

The `ScanFile` call reads 100 integers from a binary file and stores them in the integer array `readings`. If the `ScanFile` call is successful, `nbytes` = 400 (100 integers, 4 bytes per integer).

Binary File to Real Array, Assuming a Fixed Number of Elements

```
double waveform[100];
int file_handle, nbytes;
file_handle = OpenFile ("FILE.DAT", 1, 2, 0);
ScanFile (file_handle, "%100f>%100f", waveform);
nbytes = NumFmtdBytes ();
CloseFile (file_handle);
```

The `ScanFile` call reads 100 real numbers from a binary file and stores them in the real array `waveform`. If the `ScanFile` call is successful, `nbytes` = 800 (100 integers, 8 bytes per real number).

Binary File to Real Array, Assuming a Variable Number of Elements

```
void StoreArray (double x[], int count, char filename[])
{
    int file_handle;
    file_handle = OpenFile (filename, 1, 2, 0);
    ScanFile (file_handle, "%*f>%*f", count, count, x);
    CloseFile (file_handle);
}
```

This example shows how you can use a subroutine to read an array of real numbers from a binary file. The subroutine takes as parameters a real array, the number of elements to be read, and the filename.

The `ScanFile` call reads the first `count` elements of `x` from a binary file. `ScanFile` matches the two asterisks (`*`) in the format string to `count`. For instance, if `count` is 100, the format string is equivalent to `%100f>100f`.

Reading an Integer from the Standard Input

```
int n, num_readings;
n = 0;
while (n != 1) {
    FmtOut ("Enter number of readings:  ");
    n = ScanIn ("%l>%i", &num_readings);
}
```

This example shows how to get user input from the keyboard. The `FmtOut` call writes the prompt string to the screen without a linefeed or carriage return. The `ScanIn` call attempts to read an integer value from the keyboard and place it in `num_readings`. If `ScanIn` succeeds, it returns `1`, and `FmtOut` exits the loop. Otherwise, `FmtOut` repeats the prompt string.

The format string in the `ScanIn` call contains a source specifier of `%l`. This has two consequences. First, `ScanIn` returns whenever the user presses <Enter>, even if the input line is empty. This allows the prompt string to repeat at the beginning of each line until the user enters an integer value. Second, `ScanIn` discards any characters entered after the integer value.

Reading a String from the Standard Input

```
char filename[41];
int n;
n = 0;
while (n != 1) {
    FmtOut ("Enter file name:  ");
    n = ScanIn ("%l>%s[w40q]", filename);
}
```

This example is similar to the previous example except that `ScanIn` reads a string from the keyboard instead of an integer. The `w` modifier prevents `ScanIn` from writing beyond the end of `filename`. Notice that the width specified is one less than the size of `filename`. This allows room for the ASCII NUL that `ScanIn` appends at the end of `filename`. The `q` modifier causes `ScanIn` to fill any unused bytes at the end of `filename` with ASCII NULs. Without the `q` modifier, all unused bytes are filled with spaces, except for the ASCII NUL at the end.

The call to `ScanIn` in this example skips over leading spaces and tabs and terminates the string on an embedded space. For other options, refer to the *String to String* example earlier in this section.

Reading a Line from the Standard Input

```
char buf[81];
nbytes = ReadLine (0, buf, 80);
```

The previous two examples show how to read single items from the keyboard. When you are prompted to enter several items on one line, it is often easier to read the entire line into a buffer before parsing it. You can do this using `ReadLine`.

The first parameter to `ReadLine` is a file handle. In this case, the file handle is zero, which is the handle reserved for the Standard Input. The other two parameters are a buffer and the maximum number of bytes to place in the buffer. `ReadLine` always appends an ASCII NUL at the end of the bytes read. Thus, the maximum number of bytes passed to `ReadLine` must be one less than the size of the buffer.

`ReadLine` transfers every character from the input line to the buffer, including leading spaces, embedded spaces, and trailing spaces, until `ReadLine` transfers the maximum number of bytes (for example, 80). `ReadLine` discards any remaining characters at the end of the line. `ReadLine` never transfers the linefeed to the buffer.

`ReadLine` returns the number of bytes read, including the bytes discarded but excluding the linefeed.

3

Analysis Library

This chapter describes the functions in the LabWindows/CVI Analysis Library. The *Analysis Library Function Overview* section contains general information about the Analysis Library functions and panels. The *Analysis Library Function Reference* section contains an alphabetical list of the function descriptions.

Analysis Library Function Overview

The Analysis Library includes functions for 1D and 2D array manipulation, complex operations, matrix operations, and statistics. This section contains general information about the Analysis Library functions and panels.

Analysis Library Function Panels

The Analysis Library function panels are grouped in the tree structure in Table 3-1 according to the types of operations they perform.

The first- and second-level headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels. Each analysis function panel generates one analysis function call.

Table 3-1. Functions in the Analysis Library Function Tree

Class/Panel Name	Function Name
Array Operations	
1D Operations	
Clear Array	`Clear1D`
Set Array	`Set1D`
Copy Array	`Copy1D`
1D Array Addition	`Add1D`
1D Array Subtraction	`Sub1D`
1D Array Multiplication	`Mul1D`
1D Array Division	`Div1D`
1D Absolute Value	`Abs1D`
1D Negative Value	`Neg1D`
1D Linear Evaluation	`LinEv1D`
1D Maximum & Minimum	`MaxMin1D`

Table 3-1. Functions in the Analysis Library Function Tree (Continued)

Class/Panel Name	Function Name
Array Operations (continued)	
1D Operations (continued)	
1D Array Subset	`Subset1D`
1D Sort Array	`Sort`
2D Operations	
2D Array Addition	`Add2D`
2D Array Subtraction	`Sub2D`
2D Array Multiplication	`Mul2D`
2D Array Division	`Div2D`
2D Linear Evaluation	`LinEv2D`
2D Maximum & Minimum	`MaxMin2D`
Complex Operations	
Complex Numbers	
Complex Addition	`CxAdd`
Complex Subtraction	`CxSub`
Complex Multiplication	`CxMul`
Complex Division	`CxDiv`
Complex Reciprocal	`CxRecip`
Rectangular to Polar	`ToPolar`
Polar to Rectangular	`ToRect`
1D Complex Operations	
1D Complex Addition	`CxAdd1D`
1D Complex Subtraction	`CxSub1D`
1D Complex Multiplication	`CxMul1D`
1D Complex Division	`CxDiv1D`
1D Complex Linear Evaluation	`CxLinEv1D`
1D Rectangular to Polar	`ToPolar1D`
1D Polar to Rectangular	`ToRect1D`
Statistics	
Mean	`Mean`
Standard Deviation	`StdDev`
Histogram	`Histogram`
Vector & Matrix Algebra	
Dot Product	`DotProduct`
Transpose	`Transpose`
Determinant	`Determinant`
Invert Matrix	`InvMatrix`
Multiply Matrix	`MatrixMul`
Get Error String	`GetAnalysisErrorString`

Class and Subclass Descriptions

- The Array Operations function panels perform arithmetic operations on 1D and 2D arrays.
  - 1D Operations, a subclass of Array Operations, contains function panels that perform 1D array arithmetic.
  - 2D Operations, a subclass of Array Operations, contains function panels that perform 2D array arithmetic.
- The Complex Operations function panels perform complex arithmetic operations. The Complex Operations function panels can operate on complex scalars or 1D arrays. The functions process the real and imaginary parts of complex numbers separately.
  - Complex Numbers, a subclass of Complex Operations, contains function panels that perform scalar complex arithmetic.
  - 1D Complex Operations, a subclass of Complex Operations, contains function panels that perform complex arithmetic on 1D complex arrays.
- The Statistics function panels perform basic statistics functions.
- The Vector & Matrix Algebra function panels perform vector and matrix operations. Vectors and matrices are represented by 1D and 2D arrays, respectively.
- The Array Utilities function panels copy, initialize, and clear arrays.
- Miscellaneous is a class of function panels for miscellaneous Analysis Library functions.

The online help with each panel contains specific information about using each function panel.

Hints for Using Analysis Function Panels

With the analysis function panels, you can manipulate scalars and arrays of data interactively. You might find it helpful to use the Analysis Library function panels in conjunction with the User Interface Library function panels to view the results of analysis routines. When using the Analysis Library function panels, remember the following:

- The computer on which you run LabWindows/CVI affects the processing speed of the analysis functions. A numeric coprocessor, especially, increases the speed of floating-point computations. If you are using an Analysis Library function panel and nothing seems to happen for an unusually long time, remember the constraints of your hardware.
- Many analysis routines for arrays run in place. That is, the functions can store the input and output data in the same array. This point is important to keep in mind when you process large amounts of data. Large double-precision arrays consume a lot of memory. If the results you want do not require that you keep the original array or intermediate arrays of data, perform analysis operations in place where possible.
- The Interactive window maintains a record of generated code. If you forget to keep the code from a function panel, you can cut and paste code between the Interactive and Program windows.

The online help with each panel contains specific information about operating each function panel.

Reporting Analysis Errors

The functions in the Analysis Library return status information through a return value.

If the return value **status** is zero after an Analysis Library function call, the function properly executed with no errors. Otherwise, the functions set **status** to the appropriate error value. Table 3-2 at the end of this chapter lists error messages that correspond to the possible **status** values.

Analysis Library Function Reference

This section describes each function in the LabWindows/CVI Analysis Library in alphabetical order.

Abs1D

```
int status = Abs1D (double x[], int n, double y[]);
```

Purpose

Finds the absolute value of the **x** input array. `Abs1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
y	double-precision array	Absolute value of input array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Add1D

```
int status = Add1D (double x[], double y[], int n, double z[]);
```

Purpose

Adds 1D arrays. `Add1D` obtains the ith element of the output array using the following formula:

$$z_i = x_i + y_i$$

`Add1D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
y	double-precision array	Input array.
n	integer	Number of elements to add.

Output

Name	Type	Description
z	double-precision array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Add2D

```
int status = Add2D (void *x, void *y, int n, int m, void *z);
```

Purpose

Adds 2D arrays. `Add2D` obtains the $(i, j)^{th}$ element of the output array using the following formula:

$$z_{i,j} = x_{i,j} + y_{i,j}$$

`Add2D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision 2D array	Input array.
y	double-precision 2D array	Input array.
n	integer	Number of elements in first dimension.
m	integer	Number of elements in second dimension.

Output

Name	Type	Description
z	double-precision 2D array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Clear1D

```
int status = Clear1D (double x[], int n);
```

Purpose

Sets the elements of the **x** array to 0.0.

Parameters

Input

Name	Type	Description
n	integer	Number of elements in **x**.

Output

Name	Type	Description
x	double-precision array	Cleared array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Copy1D

```
int status = Copy1D (double x, int n, double y[]);
```

Purpose

Copies the elements of the **x** array. Use `Copy1D` to duplicate arrays for in place operations.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
y	double-precision array	Duplicated array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxAdd

```
int status = CxAdd (double xr, double xi, double yr, double yi, double *zr,
                    double *zi);
```

Purpose

Adds two complex numbers, *x* and *y*. `CxAdd` obtains the resulting complex number, *z*, using the following formulas:

$$zr = xr + yr$$

$$zi = xi + yi$$

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Output

Name	Type	Description
zr	double-precision pointer	Real part of *z*.
zi	double-precision pointer	Imaginary part of *z*.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxAdd1D

```
int status = CxAdd1D (double xr[], double xi[], double yr[], double yi[],
                      int n, double zr[], double zi[]);
```

Purpose

Adds two 1D complex arrays, **x** and **y**. `CxAdd1D` obtains the ith element of the resulting complex array, **z**, using the following formulas:

$$zr_i = xr_i + yr_i$$

$$zi_i = xi_i + yi_i$$

`CxAdd1D` can perform the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input

Name	Type	Description
xr	double-precision array	Real part of **x**.
xi	double-precision array	Imaginary part of **x**.
yr	double-precision array	Real part of **y**.
yi	double-precision array	Imaginary part of **y**.
n	integer	Number of elements.

Output

Name	Type	Description
zr	double-precision array	Real part of **z**.
zi	double-precision array	Imaginary part of **z**.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxDiv

```
int status = CxDiv (double xr, double xi, double yr, double yi, double *zr,
                    double *zi);
```

Purpose

Divides two complex numbers, *x* and *y*. `CxDiv` obtains the resulting complex number, *z*, using the following formulas:

$$zr = \frac{(xr \times yr + xi \times yi)}{yr^2 + yi^2}$$

$$zi = \frac{(xi \times yr - xr \times yi)}{yr^2 + yi^2}$$

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Output

Name	Type	Description
zr	double-precision	Real part of *z*.
zi	double-precision	Imaginary part of *z*.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxDiv1D

```
int status = CxDiv1D (double xr[], double xi[], double yr[], double yi[],
                      int n, double zr[], double zi[]);
```

Purpose

Divides two 1D complex arrays, **x** and **y**. `CxDiv1D` obtains the ith element of the resulting complex array, **z**, using the following formulas:

$$zr_i = \frac{(xr_i \times yr_i + xi_i \times yi_i)}{yr_i^2 + yi_i^2}$$

$$zi_i = \frac{(xi_i \times yr_i - xr_i \times yi_i)}{yr_i^2 + yi_i^2}$$

zr can be in place with **xr**; **zi** can be in place with **xi**.

Parameters

Input

Name	Type	Description
xr	double-precision array	Real part of **x**.
xi	double-precision array	Imaginary part of **x**.
yr	double-precision array	Real part of **y**.
yi	double-precision array	Imaginary part of **y**.
n	integer	Number of elements.

Output

Name	Type	Description
zr	double-precision array	Real part of **z**.
zi	double-precision array	Imaginary part of **z**.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxLinEv1D

```
int status = CxLinEv1D (double xr[], double xi[], int n, double ar,
                        double ai, double br, double bi, double yr[],
                        double yi[]);
```

Purpose

Performs a complex linear evaluation of a 1D complex array, **x** and **y**. `CxLinEv1D` obtains the ith element of the resulting complex array, **z**, using the following formulas:

$$yr_i = ar \times xr_i - ai \times xi_i + br$$

$$yi_i = ar \times xi_i + ai \times xr_i + bi$$

`CxLinEv1D` can perform the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input

Name	Type	Description
xr	double-precision array	Real part of **x**.
xi	double-precision array	Imaginary part of **x**.
n	integer	Number of elements.
ar	double-precision	Real part of *a*.
ai	double-precision	Imaginary part of *a*.
br	double-precision	Real part of *b*.
bi	double-precision	Imaginary part of *b*.

Output

Name	Type	Description
yr	double-precision array	Real part of **y**.
yi	double-precision array	Imaginary part of **y**.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxMul

```
int status = CxMul (double xr, double xi, double yr, double yi, double *zr,
                    double *zi);
```

Purpose

Multiplies two complex numbers, *x* and *y*. `CxMul` obtains the resulting complex number, *z*, using the following formulas:

$$zr = xr \times yr - xi \times yi$$

$$zi = xr \times yi + xi \times yr$$

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Output

Name	Type	Description
zr	double-precision	Real part of *z*.
zi	double-precision	Imaginary part of *z*.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxMul1D

```
int status = CxMul1D (double xr[], double xi[], double yr[], double yi[],
                      int n, double zr[], double zi[]);
```

Purpose

Multiplies two 1D complex arrays, **x** and **y**. `CxMul1D` obtains the ith element of the resulting complex array, **z**, using the following formulas:

$$zr_i = xr_i \times yr_i - xi_i \times yi_i$$

$$zi_i = xr_i \times yi_i + xi_i \times yr_i$$

`CxMul1D` can perform the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input

Name	Type	Description
xr	double-precision array	Real part of **x**.
xi	double-precision array	Imaginary part of **x**.
yr	double-precision array	Real part of **y**.
yi	double-precision array	Imaginary part of **y**.
n	integer	Number of elements.

Output

Name	Type	Description
zr	double-precision array	Real part of **z**.
zi	double-precision array	Imaginary part of **z**.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxRecip

```
int status = CxRecip (double xr, double xi, double *yr, double *yi);
```

Purpose

Finds the reciprocal of a complex number, *x*. `CxRecip` obtains the resulting complex number, *y*, using the following formulas:

$$yr = \frac{xr}{xr^2 + xi^2}$$

$$yi = \frac{-xi}{xr^2 + xi^2}$$

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.

Output

Name	Type	Description
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxSub

```
int status = CxSub (double xr, double xi, double yr, double yi, double *zr,
                    double *zi);
```

Purpose

Subtracts two complex numbers, *x* and *y*. The resulting complex number, *z*, is obtained using the following formulas:

$$zr = xr - yr$$

$$zi = xi - yi$$

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Output

Name	Type	Description
zr	double-precision	Real part of *z*.
zi	double-precision	Imaginary part of *z*.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

CxSub1D

```
int status = CxSub1D (double xr[], double xi[], double yr[], double yi[],
                          int n, double zr[], double zi[]);
```

Purpose

Subtracts two 1D complex arrays, **x** and **y**. `CxSub1D` obtains the ith element of the resulting complex array, **z**, using the formulas:

$$zr_i = xr_i - yr_i$$

$$zi_i = xi_i - yi_i$$

`CxSub1D` can perform the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input

Name	Type	Description
xr	double-precision array	Real part of **x**.
xi	double-precision array	Imaginary part of **x**.
yr	double-precision array	Real part of **y**.
yi	double-precision array	Imaginary part of **y**.
n	integer	Number of elements.

Output

Name	Type	Description
zr	double-precision array	Real part of **z**.
zi	double-precision array	Imaginary part of **z**.

Return Value

Name	Type	Description
status	integer	Refer toTable 3-2 for error codes.

Determinant

```
int status = Determinant (void *x, int n, double *det);
```

Purpose

Finds the determinant of an **n**-by-**n** 2D input matrix.

Parameters

Input

Name	Type	Description
x	double-precision 2D array	Input matrix.
n	integer	Dimension size of input matrix.

Output

Name	Type	Description
det	double-precision	Determinant.

☞ **Note** ***The input matrix must be an* n*-by-*n *square matrix.***

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Div1D

```
int status = Div1D (double x[], double y[], int n, double z[]);
```

Purpose

Divides two 1D arrays, **x** and **y**. `Div1D` obtains the ith element of the output array, **z**, using the following formula:

$$z_i = \frac{x_i}{y_i}$$

`Div1D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	**x** input array.
y	double-precision array	**y** input array.
n	integer	Number of elements to divide.

Output

Name	Type	Description
z	double-precision array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Div2D

```
int status = Div2D (void *x, void *y, int n, int m, void *z);
```

Purpose

Divides two 2D arrays. `Div2D` obtains the $(i, j)^{th}$ element of the output array using the following formula:

$$z_{i,j} = \frac{x_{i,j}}{y_{i,j}}$$

`Div2D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision 2D array	**x** input array.
y	double-precision 2D array	**y** input array.
n	integer	Number of elements in first dimension.
m	integer	Number of elements in second dimension.

Output

Name	Type	Description
z	double-precision 2D array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

DotProduct

```
int status = DotProduct (double x[], double y, int n, double *dotProd);
```

Purpose

Calculates the dot product of the **x** and **y** input arrays. `DotProduct` obtains the dot product using the following formula:

$$dotProd = x \bullet y = \sum_{i=0}^{n-1} x_i \times y_i$$

Parameters

Input

Name	Type	Description
x	double-precision array	**x** input vector.
y	double-precision array	**y** input vector.
n	integer	Number of elements.

Output

Name	Type	Description
dotProd	double-precision	Dot product.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

GetAnalysisErrorString

```
char *message = GetAnalysisErrorString (int errorNum)
```

Purpose

Converts the error number an Analysis Library function returns into a meaningful error message.

Parameter

Input

Name	Type	Description
errorNum	integer	Status an Analysis Library function returns.

Return Value

Name	Type	Description
message	string	Explanation of error.

Histogram

```
int status = Histogram (double inputArray[], int numberOfElements,
                        double base, double top, int histogramArray[],
                        double axisArray[], int intervals);
```

Purpose

Calculates the histogram of the **inputArray**. If the input sequence is

$$X = \{0, 1, 3, 3, 4, 4, 4, 5, 5, 8\}$$

the Histogram: $h(X)$ of X for eight **intervals** is

$$h(x) = \{h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7\} = \{1, 1, 0, 2, 3, 2, 0, 1\}$$

Notice that the histogram of the input sequence X is a function of X.

The function obtains Histogram: $h(X)$ as follows: `Histogram` scans the input sequence X to determine the range of values in it. Then the function 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
m is the specified number of **intervals**

Let χ represent the output sequence X because the histogram is a function of X. The function evaluates elements of χ using

$$\chi_i = min + 0.5 \times \Delta x + i \times \Delta x \quad \text{for } i = 0, 1, 2, \ldots, m-1$$

`Histogram` defines the ith interval Δ_i to be the range of values from $\chi_i - 0.5 \times \Delta x$ up to but not including $\chi_i + 0.5 \times \Delta x$

$$\Delta_i = [\chi_i - 0.5 \times \Delta x : \chi_i + 0.5 \times \Delta x) \quad \text{for } i = 0, 1, 2, \ldots, m-1$$

and defines the function $y_i(x)$ to be

$$y_i(x) = \begin{cases} 1 & \text{if } x \in \text{ union of } \Delta_i \\ 0 & \text{elsewhere} \end{cases}$$

`Histogram` has unity value if the value of x falls within the specified interval. Otherwise it is zero. Notice that the interval Δ_i is centered about χ_i, and its width is Δ_x.

The last interval, Δ_{m-1}, is defined as $[\chi_{m-i} - 0.5 \times \Delta x : \chi_{m-i} + 0.5 \times \Delta x]$. In other words, if a value equals *max*, it is counted as belonging to the last interval.

Finally, `Histogram` evaluates the histogram sequence h using

$$h_i = \sum_{j=0}^{n-1} y_i(x_j) \quad \text{for } i = 0, 1, 2, \ldots, m-1$$

where h_i represents the elements of the output sequence Histogram: $h(X)$
n is the number of elements in the input sequence X

`Histogram` obtains the histogram by counting the number of times the elements in the input array fall in the ith interval.

Parameters

Input

Name	Type	Description
inputArray	double-precision array	Input array.
numberOfElements	integer	Number of elements in **inputArray**.
base	double-precision	Lower range.
top	double-precision	Upper range.
intervals	integer	Number of intervals.

Output

Name	Type	Description
histogramArray	integer array	Histogram of **inputArray**.
axisArray	double-precision array	Histogram axis array; contains the midpoint values of the intervals.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

InvMatrix

```
int status = InvMatrix (void *x, int n, void *y);
```

Purpose

Finds the inverse matrix of an input matrix. `InvMatrix` can perform the operation in place; that is, **x** and **y** can be the same matrices.

Parameters

Input

Name	Type	Description
x	double-precision 2D array	Input matrix.
n	integer	Dimension size of matrix.

Output

Name	Type	Description
y	double-precision 2D array	Inverse matrix.

Note ***The input matrix must be an* n*-by-*n *square matrix.***

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

LinEv1D

```
int status = LinEv1D (double x[], int n, double a, double b, double y[]);
```

Purpose

Performs a linear evaluation of a 1D array, **x**. `LinEv1D` obtains the ith element of the output array, **y**, using the formula:

$$y_i = a \times x_i + b$$

`LinEv1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.
a	double-precision	Multiplicative constant.
b	double-precision	Additive constant.

Output

Name	Type	Description
y	double-precision array	Linearly evaluated array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

LinEv2D

```
int status = LinEv2D (void *x, int n, int m, double a, double b, void *y);
```

Purpose

Performs a linear evaluation of a 2D array, **x**. `LinEv2D` obtains the $(i, j)^{th}$ element of the output array, **y**, using the formula:

$$y_{i,j} = a \times x_{i,j} + b$$

`LinEv2D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision 2D array	Input array.
n	integer	Number of elements in first dimension.
m	integer	Number of elements in second dimension.
a	double-precision	Multiplicative constant.
b	double-precision	Additive constant.

Output

Name	Type	Description
y	double-precision 2D array	Linearly evaluated array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

MatrixMul

```
int status = MatrixMul (void *X, void *Y, int n, int k, int m, void *Z);
```

Purpose

Multiplies two 2D input matrices, **X** and **Y**. `MatrixMul` obtains the $(i, j)^{th}$ element of the output matrix, **Z**, using the formula:

$$Z_{i,j} = \sum_{p=0}^{k-1} x_{i,p} \times y_{p,j}$$

Parameters

Input

Name	Type	Description
X	double-precision 2D array	**X** input matrix.
Y	double-precision 2D array	**Y** input matrix.
n	integer	First dimension of **X**.
k	integer	Second dimension of **X**; first dimension of **Y**.
m	integer	Second dimension of **Y**.

Output

Name	Type	Description
Z	double-precision 2D array	Output matrix.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Parameter Discussion

Confirm that the array sizes are correct. You must meet the following array sizes:

- **X** must be **n** by **k**.
- **Y** must be **k** by **m**.
- **Z** must be **n** by **m**.

Example

```
/* Multiply two matrices. Note: A x B - B x A, in general. */
double  x[10][20], y[20][15], z[10][15];
int n, k, m;
n = 10;
k = 20;
m = 15;
MatrixMul (x, y, n, k, m, z);
```

MaxMin1D

```
int status = MaxMin1D (double x[], int n, double *max, int *imax, double *min,
                        int *imin);
```

Purpose

Finds the maximum and minimum values in the input array and the respective indices of the first occurrence of the maximum and minimum values.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
max	double-precision	Maximum value.
imax	integer	Index of **max** in **x** array.
min	double-precision	Minimum value.
imin	integer	Index of **min** in **x** array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Example

```
/* Generate an array with random and find the maximum and minimum
values. */
double  x[20], y[20];
double  max, min;
int  n, imax, imin;
n = 20;
Uniform (n, 17, x);
MaxMin1D (x, n, &max, &imax, &min, &imin);
```

MaxMin2D

```
int status = MaxMin2D (void *X, int n, int m, double *max, int *imax,
                           int *jmax, double *min, int *imin, int *jmin);
```

Purpose

Finds the maximum and the minimum values in the 2D input array and the respective indices of the first occurrence of the maximum and minimum values. `MaxMin2D` scans the **X** array by rows.

Parameters

Input

Name	Type	Description
X	double-precision 2D array	Input array.
n	integer	Number of elements in first dimension of **X**.
m	integer	Number of elements in second dimension of **X**.

Output

Name	Type	Description
max	double-precision	Maximum value.
imax	integer	Index of **max** in **X** array (first dimension).
jmax	integer	Index of **max** in **X** array (second dimension).
min	double-precision	Minimum value.
imin	integer	Index of **min** in **X** array (first dimension).
jmin	integer	Index of **min** in **X** array (second dimension).

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Example

```
/* This example finds the maximum and minimum values as well as their
location within the array. */
double x[5][10], max, min;
int n, m, imax, jmax, imin, jmin;
n = 5;
m = 10;
MaxMin2D (x, n, m, &max, &imax, &jmax, &min, &imin, &jmin);
```

Mean

```
int status = Mean (double x[], int n, double *meanval);
```

Purpose

Calculates the mean, or average, value of the input array. `Mean` calculates the mean using the following formula:

$$meanval = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
meanval	double-precision	Mean value.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Mul1D

```
int status = Mul1D (double x[], double y[], int n, double z[]);
```

Purpose

Multiplies two 1D arrays. `Mul1D` obtains the ith element of the output array using the following formula:

$$z_i = x_i \times y_i$$

`Mul1D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	**x** input array.
y	double-precision array	**y** input array.
n	integer	Number of elements to multiply.

Output

Name	Type	Description
z	double-precision array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Mul2D

```
int status = Mul2D (void *X, void *Y, int n, int m, void *Z);
```

Purpose

Multiplies two 2D arrays, **X** and **Y**. `Mul2D` obtains the $(i, j)^{th}$ element of the output array, **Z**, using the following formula:

$$z_{i,j} = x_{i,j} + y_{i,j}$$

`Mul2D` can perform the operation in place; that is, **Z** can be the same array as either **X** or **Y**.

Parameters

Input

Name	Type	Description
X	double-precision 2D array	**X** input array.
Y	double-precision 2D array	**Y** input array.
n	integer	Number of elements in first dimension.
m	integer	Number of elements in second dimension.

Output

Name	Type	Description
Z	double-precision 2D array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Neg1D

```
int status = Neg1D (double x[], int n, double y[]);
```

Purpose

Negates the elements of the input array. `Neg1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
y	double-precision array	Negated values of the **x** input array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Set1D

```
int status = Set1D (double x[], int n, double a);
```

Purpose

Sets the elements of the **x** array to a constant value.

Parameters

Input

Name	Type	Description
n	integer	Number of elements in **x**.
a	double-precision	Constant value.

Output

Name	Type	Description
x	double-precision array	Result array; set to the value of **a**.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Sort

```
int status = Sort (double x[], int n, int direction, double y[]);
```

Purpose

Sorts the **x** input array in ascending or descending order. `Sort` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements to sort.
direction	integer	`0` = ascending nonzero = descending

Output

Name	Type	Description
y	double-precision array	Sorted array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Example

```
/* Generate a random array of numbers and sort them in ascending
order. */
double  x[200], y[200];
int  n;
int  dir;
n = 200;
dir = 0;
Uniform (n, 17, x);
Sort (x, n, dir, y);
```

StdDev

```
int status = StdDev (double x[], int n, double *meanval, double *sDev);
```

Purpose

Calculates the standard deviation and the mean, or average, values of the input array. `StdDev` uses the following formulas to find the mean and the standard deviation:

$$meanval = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

$$sDev = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i - meanval)^2}{n}}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
meanval	double-precision	Mean value.
sDev	double-precision	Standard deviation.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Sub1D

```
int status = Sub1D (double x[], double y[], int n, double z[]);
```

Purpose

Subtracts two 1D arrays. `Sub1D` can obtain the ith element of the output array using the following formula:

$$z_i = x_i - y_i$$

`Sub1D` can perform the operation in place; that is, **z** can be either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	**x** input array.
y	double-precision array	**y** input array.
n	integer	Number of elements to subtract.

Output

Name	Type	Description
z	double-precision array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Sub2D

```
int status = Sub2D (void *X, void *Y, int n, int m, void *Z);
```

Purpose

Subtracts two 2D arrays. `Sub2D` obtains the $(i, j)^{th}$ element of the output array using the formula:

$$z_{i,j} = x_{i,j} - y_{i,j}$$

`Sub2D` can perform the operation in place; that is, **Z** can be either **X** or **Y**.

Parameters

Input

Name	Type	Description
X	double-precision 2D array	**X** input array.
Y	double-precision 2D array	**Y** input array.
n	integer	Number of elements in first dimension.
m	integer	Number of elements in second dimension.

Output

Name	Type	Description
Z	double-precision 2D array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Subset1D

```
int status = Subset1D (double x[], int n, int index, int length, double y[]);
```

Purpose

Extracts a subset of the input array. The output array contains the number of elements you specify by the **length**. `Subset1D` starts copying from **x** to **y** at the **index** element of **x**.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.
index	integer	Initial **index** for the subset.
length	integer	Number of elements to copy to the subset.

Output

Name	Type	Description
y	double-precision array	Subset array.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Example

```
/* The following example generates  y = {0.0, 1.0, 2.0, 3.0}. */
double  x[11], y[4], first, last;
int  n, index, length;
n = 11;
index = 5;
length = 4;
first = -5.0;
last = 5.0;
Ramp (n, first, last, x);
Subset1D (x, n, index, length, y);
```

ToPolar

```
int status = ToPolar (double x, double y, double *mag, double *phase);
```

Purpose

Converts the rectangular coordinates (**x**, **y**) to polar coordinates (**mag**, **phase**). `ToPolar` obtains the polar coordinates using the following formulas:

$$mag = \sqrt{x^2 + y^2}$$

$$phase = \arctan\left(\frac{y}{x}\right)$$

The **phase** value is in the range $[-\pi : \pi]$.

Parameters

Input

Name	Type	Description
x	double-precision	**x** coordinate.
y	double-precision	**y** coordinate.

Output

Name	Type	Description
mag	double-precision	Magnitude.
phase	double-precision	Phase, in radians.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Example

```
/* Convert the rectangular coordinates to polar coordinates. */
double  x, y, mag, phase;
x = 1.5;
y = -2.5;
ToPolar (x, y, &mag, &phase);
```

ToPolar1D

```
int status = ToPolar1D (double x[], double y[], int n, double mag[],
                        double phase[]);
```

Purpose

Converts the set of rectangular coordinate points (**x**, **y**) to a set of polar coordinate points (**mag**, **phase**). `ToPolar1D` obtains the ith element of the polar coordinate set using the following formulas:

$$mag_i = \sqrt{x_i^2 + y_i^2}$$

$$phase_i = \arctan\left(\frac{y_i}{x_i}\right)$$

The **phase** value is in the range $[-\pi : \pi]$.

`ToPolar1D` can perform the operations in place; that is, **x** and **mag**, and **y** and **phase**, can be the same arrays, respectively.

Parameters

Input

Name	Type	Description
x	double-precision array	**x** coordinate.
y	double-precision array	**y** coordinate.
n	integer	Number of elements.

Output

Name	Type	Description
mag	double-precision array	Magnitude.
phase	double-precision array	Phase, in radians.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

ToRect

```
int status = ToRect (double mag, double phase, double *x, double *y);
```

Purpose

Converts the polar coordinates (**mag**, **phase**) to rectangular coordinates (**x**, **y**). `ToRect` obtains the rectangular coordinates using the following formulas:

$$x = mag \times \cos(phase)$$

$$y = mag \times \sin(phase)$$

Parameters

Input

Name	Type	Description
mag	double-precision	Magnitude.
phase	double-precision	Phase, in radians.

Output

Name	Type	Description
x	double-precision	**x** coordinate.
y	double-precision	**y** coordinate.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

ToRect1D

```
int status = ToRect1D (double mag[], double phase[], int n, double x[],
                       double y[]);
```

Purpose

Converts the set of polar coordinate points (**mag**, **phase**) to a set of rectangular coordinate points (**x**, **y**). `ToRect1D` obtains the ith element of the rectangular set using the following formulas:

$$x_i = mag_i \times \cos(phase_i)$$

$$y_i = mag_i \times \sin(phase_i)$$

`ToRect1D` can perform the operations in place; that is, **x** and **mag**, and **y** and **phase**, can be the same arrays, respectively.

Parameters

Input

Name	Type	Description
mag	double-precision array	Magnitude.
phase	double-precision array	Phase, in radians.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	**x** coordinate.
y	double-precision array	**y** coordinate.

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Transpose

```
int status = Transpose (void *x, int n, int m, void *y);
```

Purpose

Finds the transpose of a 2D input matrix. `Transpose` obtains the $(i, j)^{th}$ element of the resulting matrix using the following formula:

$$y_{i,j} = x_{j,i}$$

Parameters

Input

Name	Type	Description
x	double-precision 2D array	Input matrix.
n	integer	Size of first dimension.
m	integer	Size of second dimension.

Output

Name	Type	Description
y	double-precision 2D array	Transpose matrix.

Note ***If the input matrix has* n*-by-*m *dimensions, the output matrix must have* m*-by-*n *dimensions.***

Return Value

Name	Type	Description
status	integer	Refer to Table 3-2 for error codes.

Error Conditions

If an error condition occurs during a call to any of the functions in the Analysis Library, the status return value contains the error code. This code is a value that specifies the type of error that occurred. Table 3-2 shows the currently defined error codes and the associated meanings.

Table 3-2. Analysis Library Error Codes

Symbolic Name	Code	Error Message
`BaseGETopAnlysErr`	`-20101`	Base must be less than Top.
`DivByZeroAnlysErr`	`-20060`	Divide by zero error.
`IndexLengthAnlysErr`	`-20018`	The following condition must be met: $0 \leq (\text{index} + \text{length}) < \text{samples}$.
`NoAnlysErr`	`0`	No error; the call was successful.
`OutOfMemAnlysErr`	`-20001`	There is not enough space left to perform the specified routine.
`SamplesGEZeroAnlysErr`	`-20004`	The number of samples must be greater than or equal to zero.
`SamplesGTZeroAnlysErr`	`-20003`	The number of samples must be greater than zero.
`SingularMatrixAnlysErr`	`-20041`	The input matrix is singular. The system of equations cannot be solved.

4

GPIB/GPIB-488.2 Library

This chapter describes the functions in the LabWindows/CVI GPIB Library. The *GPIB Library Function Overview* section contains general information about the GPIB Library functions and panels, the GPIB DLL, and guidelines and restrictions you should know when using the GPIB Library. The *GPIB Function Reference* section contains an alphabetical list of descriptions for the Device Manager functions, the callback installation functions, and the functions for returning the thread-specific status variables. Refer to your NI-488.2 or NI-488.2M function reference for detailed descriptions of the NI-488 and NI-488.2 functions.

GPIB Library Function Overview

This section describes the functions in the LabWindows/CVI GPIB Library. These functions are arranged alphabetically according to their names in C. For detailed function descriptions, refer to the NI-488.2 function reference manual that accompanied your GPIB interface.

GPIB Functions Library Function Panels

The GPIB Functions Library function panels are grouped in the tree structure in Table 4-1 according to the types of operations they perform.

The first- and second-level headings in the function tree are names of the function classes. Function classes are groups of related function panels. The third-level headings are the names of individual function panels. Each GPIB function panel generates a GPIB function call.

Table 4-1. Functions in the GPIB/GPIB-488.2 Library Function Tree

Class/Panel Name	Function Name
Open/Close	
Open Device	`OpenDev`
Close Device	`CloseDev`
Close Instrument Devices	`CloseInstrDevs`
Find Board/Device	`ibfind`
Find Unused Device	`ibdev`
Offline/Online	`ibonl`
Configuration	
Change Primary Address	`ibpad`
Change Secondary Address	`ibsad`
Change Access Board	`ibbna`
Change Time Out Limit	`ibtmo`

Table 4-1. Functions in the GPIB/GPIB-488.2 Library Function Tree (Continued)

Class/Panel Name	Function Name
Configuration (continued)	
Set EOS Character	`ibeos`
Enable/Disable END	`ibeot`
Enable/Disable DMA	`ibdma`
System Control	`ibrsc`
Change Config Parameter	`ibconfig`
Get Config Parameter	`ibask`
I/O	
Read	`ibrd`
Read Asynchronously	`ibrda`
Read to File	`ibrdf`
Write	`ibwrt`
Write Asynchronously	`ibwrta`
Write from File	`ibwrtf`
Stop Asynchronous I/O	`ibstop`
Device Control	
Get Serial Poll Byte	`ibrsp`
Clear Device	`ibclr`
Trigger device	`ibtrg`
Check for Listeners	`ibln`
Wait for Event (Dev)	`ibwait`
Go to Local (Dev)	`ibloc`
Parallel Poll Cfg (Dev)	`ibppc`
Pass Control	`ibpct`
Bus Control	
Send Interface Clear	`ibsic`
Become Active Controller	`ibcac`
Go to Standby	`ibgts`
Set/Clear Remote Enable	`ibsre`
Send Commands	`ibcmd`
Send Commands (Async)	`ibcmda`
Parallel Poll	`ibrpp`
Read Control Lines	`iblines`
Board Control	
Wait for Board Event	`ibwait`
Dequeue Board Event	`ibevent`
Set UNIX Signal Request	`ibsgnal`
Go to Local Mode	`ibloc`
Parallel Poll Configuration	`ibppc`
Request Service	`ibrsv`
Set/Clear IST	`ibist`
Write to Board Key	`ibwrtkey`
Read from Board Key	`ibrdkey`

Table 4-1. Functions in the GPIB/GPIB-488.2 Library Function Tree (Continued)

Class/Panel Name	Function Name
Callbacks (Windows only)	
Install Synchronous Callback	`ibInstallCallback`
Install Asynchronous Callback	`ibnotify`
Locking (GPIB-ENET Only)	
Lock	`iblock`
Unlock	`ibunlock`
Thread-Specific Status	
Get Ibsta for Thread	`ThreadIbsta`
Get Iberr for Thread	`ThreadIberr`
Get Ibcnt for Thread	`ThreadIbcnt`
Get Ibcntl for Thread	`ThreadIbcntl`
GPIB-488.2 Functions	
Device I/O	
Send	`Send`
Send to Multiple Device	`SendList`
Receive	`Receive`
Trigger and Clear	
Trigger Device	`Trigger`
Trigger Multiple Devices	`TriggerList`
Clear Device	`DevClear`
Clear Multiple Devices	`DevClearList`
SRQ and Serial Polls	
Test SRQ Line	`TestSRQ`
Wait for SRQ	`WaitSRQ`
Find Requesting Device	`FindRQS`
Read Status Byte	`ReadStatusByte`
Serial Poll All Devices	`AllSpoll`
Parallel Polls	
Parallel Poll	`PPoll`
Parallel Poll Config	`PPollConfig`
Parallel Poll Unconfig	`PPollUnconfig`
Remote/Local	
Enable Remote Operation	`EnableRemote`
Enable Local Operation	`EnableLocal`
Set Remote with Lockout	`SetRWLS`
Send Local Lockout	`SendLLO`
System Control	
Reset System	`ResetSys`
Send Interface Clear	`SendIFC`
Conduct Self-Tests	`TestSys`
Find All Listeners	`FinsLstn`
Pass Control	`PassControl`
Low-Level I/O	
Send Commands	`SendCmds`
Setup for Sending	`SendSetup`

Table 4-1. Functions in the GPIB/GPIB-488.2 Library Function Tree (Continued)

Class/Panel Name	Function Name
GPIB-488.2 Functions (continued)	
Low-Level I/O (continued)	
Send Data Bytes	`SendDataBytes`
Setup for Receiving	`ReceiveSetup`
Receive Response Message	`RcvRespMsg`

Class and Subclass Descriptions

- The Open/Close function panels open and close GPIB interfaces and devices.
- The Configuration function panels alter configuration parameters you set during installation of the GPIB handler or during the execution of previous program statements.
- The I/O function panels read and write data over the GPIB. You can use these functions at the board or the device level.
- The Device Control function panels provide high-level, commonly-used GPIB services for instrument control applications.
- The Bus Control function panels provide low-level control of the GPIB bus.
- The Board Control function panels provide low-level control of the GPIB interface. Normally, you use these functions when the GPIB interface is not Controller-In-Charge.
- The Callbacks function panels install callback functions to invoke when certain GPIB events occur. The functions in this class are available only under Windows. Under UNIX, you can use the `ibsgnl` function.
- The Locking function panels allow you to control simultaneous access to the GPIB-ENET from multiple applications or computers.
- The Thread-Specific Status function panels return the value of the thread-specific GPIB status variables for the current thread. The functions in this class are necessary only for multithreaded applications. They are available only on Windows 95/NT.
- The GPIB 488.2 Functions function panels directly adhere to the IEEE-488.2 standard for communicating with and controlling GPIB devices.
  - The Device I/O function panels read data from and write data to devices on the GPIB.
  - The Trigger and Clear function panels trigger and clear GPIB devices.
  - The SRQ and Serial Polls function panels handle service requests and perform serial polls.
  - The Parallel Polls function panels conduct parallel polls and configure devices to respond to them.
  - The Remote/Local function panels enable and disable operation of devices remotely through the GPIB or locally through the front panel of the device.

- The System Control function panels perform system-wide functions, obtain system-wide status information, and pass system control to other devices.
- The Low-Level I/O function panels perform I/O functions at a lower-level than the function panels in the other classes.

The online help with each panel contains specific information about operating each function panel.

GPIB Library Concepts

This section contains general information about the GPIB Library, the GPIB device driver, guidelines and restrictions you should know when using the GPIB Library, and descriptions of the types of GPIB functions the GPIB Library contains.

GPIB Libraries and the GPIB Dynamic Link Library/Device Driver

LabWindows/CVI for Windows uses the National Instruments standard GPIB DLL for Windows. LabWindows/CVI for Sun uses the standard Sun Solaris-installed GPIB device drivers. These drivers are packaged with your GPIB interface and are not included with LabWindows/CVI. LabWindows/CVI does not require any special procedures for installing and using the device driver. Follow the directions outlined in your interface documentation.

You can use a software utility called `ibconf`, included with your GPIB software, to specify configuration parameters for devices on the GPIB. If your device has special configuration parameters, such as a secondary address or a special termination character, you can specify these using `ibconf`. When you use the LabWindows/CVI GPIB Library function panels, parameters you specified using `ibconf` are still in effect. You can modify configuration parameters directly from one of the LabWindows/CVI configuration function panels or from your program.

If you use a LabWindows/CVI Instrument Driver module, you do not need to make any changes using `ibconf`. The module takes into account any special configuration requirements for the instrument. If special parameters must be specified, the module sets them programmatically.

Guidelines and Restrictions for Using the GPIB Libraries

Follow these guidelines when using the GPIB Libraries:

- Before performing any other operations, open the device. You must use `OpenDev`, `ibfind`, or `ibdev`. Instrument modules must use `OpenDev`. When you open a device, an integer value that represents a device descriptor is returned. All subsequent operations that involve a particular device require that you specify this device descriptor.
- If you use `OpenDev`, you should use `CloseDev` to close the device at the end of the program.
- Each GPIB Library function panel has three global controls labeled Status, Error, and Count. These controls show the values of the GPIB status (`ibsta`), error (`iberr`), and byte count (`ibcntl`) variables.
  - The Status control displays in hexadecimal format. The help information for Status explains the meaning of each bit in the status word. If the most significant bit is set, a GPIB error has occurred.
  - When an error occurs, the Error control displays an error number. The help information for Error describes the type of error associated with each error number.
  - Count displays the number of bytes transferred over the GPIB during the most recent bus transfer.

Note ***When writing instrument modules, you must use the Device Manager functions (*`OpenDev` *and* `CloseDev`*) instead of* `ibfind` *or* `ibdev`*. You also must use the Device Manager functions in application programs that make calls to instrument modules. The Device Manager functions allow instrument modules to open devices without specific device names, thereby preventing device name conflicts. They also help the LabWindows/CVI interactive program ensure that devices are closed when no longer needed.***

Device and Board Functions

Device functions are high-level functions that execute command sequences to handle bus management operations required for operations such as reading from and writing to devices or polling them for status. Device functions access a specific device and handle the addressing and bus management protocol for that device. You do not need to know any GPIB protocol or bus management details. A descriptor of the accessed device is one of the arguments of the function.

In contrast, board functions are low-level functions that perform rudimentary GPIB operations. They are necessary because high-level functions might not always meet the requirements of applications. In such cases, low-level functions offer the flexibility to meet your application needs.

Board functions access the GPIB interface directly and require you to configure the addressing and bus management protocol for the bus. A descriptor of the accessed board is one of the arguments of the function.

Automatic Serial Polling

Automatic Serial Polling relieves you of the burden of sorting out occurrences of SRQ and obtaining status bytes from devices. To enable Automatic Serial Polling, (or *autopolling*), use the configuration utility `ibconf` or the configuration function `ibconfig`. If you enable autopolling, the handler automatically conducts serial polls when SRQ is asserted.

As part of the autopoll procedure, the handler stores each positive serial poll response in a queue associated with each device. A positive response has the RQS or hex 40 bit set in the device status byte. The handler stores the positive responses in a queue because some devices can send multiple positive status bytes before your program can act on them. When the handler receives a positive response from a device, the handler sets the RQS bit of the status word (`ibsta`). The polling continues until the device unasserts SRQ or the handler detects an error condition.

If the handler cannot locate the device that requests service because no known device responds positively to the poll, or if SRQ is stuck in the asserted state because of a faulty instrument or cable, a GPIB system error exists that interferes with the proper evaluation of the RQS bit in the status words of devices. The handler reports the ESRQ error to you when you issue an `ibwait` call with the RQS bit included in the wait mask. Aside from the difficulty ESRQ causes in waiting for RQS, the error has no detrimental effects on other GPIB operations.

If you call the serial poll function `ibrsp` and have received one or more responses previously through the automatic serial poll feature, the `ibrsp` function returns the first queued response. Other responses are read in FIFO (first-in-first-out) fashion. If the RQS bit of the status word is not set when you call `ibrsp`, the function conducts a serial poll and returns whatever response it receives.

If your application requires that requests for service be noticed, call the `ibrsp` function whenever the RQS bit appears in the status word. The serial poll response queue of a device can overflow with old status bytes when you neglect to call `ibrsp`. `ibrsp` returns the error condition ESTB when status bytes are discarded because the queue is full. If your application has no interest in SRQ or status bytes, you can ignore the occurrence of the automatic polls.

☞ **Note** ***If the RQS bit of the device status word remains set after you call*** `ibrsp`***, the response byte queue has at least one more response byte remaining. You should continue to call*** `ibrsp` ***until RQS clears.***

Autopolling Compatibility

You cannot detect the SRQI bit in device status words (`ibsta`) if you enable autopolling. The goal of autopolling is to remove the SRQ from the IEEE 488 bus, thus preventing visibility of the SRQI bit in status words for board calls and device calls. If you choose to look for SRQI in your program, you must disable autopolling.

Board functions also are incompatible with autopolling. The handler disables autopolling whenever you make a board call and re-enables it at the end of a subsequent device call.

Hardware Interrupts and Autopolling

If you disable the interrupts of the GPIB interface using `ibconf` or the `ibconfig` function, the handler detects SRQ only during calls to the handler, and autopolling can occur only at the following events:

- During a device `ibwait` for RQS
- Immediately after a device function completes and prepares to return to the application program.

If you enable hardware interrupts, the handler can respond to SRQI interrupts and perform autopolling even when the handler is not performing a function. However, the handler does not conduct an autopoll if any of the following conditions exist:

- The last GPIB call was a board call. The handler reinstates autopolling after a subsequent device call.
- GPIB I/O is in progress. In particular, during asynchronous GPIB I/O, autopolling does not occur until the asynchronous I/O completes.
- The stuck SRQ condition exists.
- You disabled autopolling by using `ibconf` or `ibconfig`.

Read and Write Termination

The IEEE 488 specification defines two methods of identifying the last byte of device-dependent (data) messages. The two methods permit a talker to send data messages of any length without the listener(s) knowing in advance the number of bytes in the transmission. The two methods are as follows:

- END message. The talker asserts the EOI (End Or Identify) signal simultaneously with transmission of the last data byte. By design, the listener stops reading when it detects a data message accompanied by EOI, regardless of the value of the byte.
- End of String (EOS) character. The talker uses a special character at the end of its data string. You can configure the listener to stop receiving data when it detects that character. You can use either a 7-bit ASCII character or a full 8-bit binary byte.

You can use these methods individually or in combination. However, you must properly configure the listener to unambiguously detect the end of a transmission.

Using the `ibconf` configuration program, you can accommodate all permissible forms of read and write termination. You can change the default configuration settings for read and write termination at run time using `ibeos` and `ibeot`. In accordance with the IEEE 488 specification, you cannot force the handler to ignore END on read operations.

Timeouts

A timeout mechanism regulates the GPIB routines that transfer command sequences or data messages. All I/O must complete within the timeout period to avoid a timeout error. The handler uses a default timeout period of 10 seconds. You can change the default timeout value with `ibconf`. In addition, you can use the function `ibtmo` to programmatically alter the timeout period.

Regardless of the I/O and wait timeout period, GPIB enforces a much shorter timeout for reading responses to serial polls. This shorter timeout period takes effect whenever you conduct a serial poll. Because devices normally respond quickly to polls, you do not need to wait the relatively lengthy I/O timeout period for a nonresponsive device.

Global Variables for the GPIB Library

The GPIB Library and the GPIB-488.2 Library use the following global variables:

- Status Word (`ibsta`)
- Error (`ibcnt`, `ibcntl`)

These variables update after each NI-488 or NI-488.2 routine to reflect the status of the device or board just accessed. Refer to your NI-488.2 user manual for detailed information on the GPIB global variables.

Multithreading under Windows 95/NT

If you use multithreading in a standalone executable, you can call GPIB functions from more than one thread at the same time under Windows 95/NT. To be truly multithreaded safe, you must use one of the following versions of the NI-488.2M driver:

- For Windows 95: Version 1.1 or later
- For Windows NT: Version 1.2 or later

Although previous versions of the drivers support multithreading, they do not support `ThreadIbsta`, `ThreadIberr`, `ThreadIbcnt`, or `ThreadIbcntl`. You need these functions to obtain thread-specific status values when calling GPIB functions from more than one thread. The global status variables `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are not reliable in this case because the GPIB Library maintains them on a *per process* basis.

Notification of SRQ and Other GPIB Events under Windows

Synchronous Callbacks

Under Windows 3.1, you can use `ibInstallCallback` to specify a function to call when an SRQ is asserted on the GPIB or when an asynchronous I/O operation completes. It is a board-level function only.

If you use Windows 95/NT, you can use `ibInstallCallback` to specify functions to invoke on the occurrence of any board-level or device-level condition on which you can wait using `ibwait`.

Callback functions you install with `ibInstallCallback` are *synchronous* callbacks; that is, LabWindows/CVI can invoke them only when it processes events. LabWindows/CVI processes events when you call `ProcessSystemEvents` or `GetUserEvent` or when `RunUserInterface` is active and you are not in a callback function. Consequently, the latency between the occurrence of the GPIB event and the invocation of the callback can be large. On the other hand, you are not restricted in what you can do in the callback function.

Asynchronous Callbacks

If you use Windows 95/NT, you can use `ibnotify` to install *asynchronous* callbacks. LabWindows/CVI can call your asynchronous callbacks at any time with respect to the rest of your program. Consequently, the latency between the occurrence of the GPIB event and the invocation of the callback is smaller than with synchronous callbacks, but you are restricted in what you can do in the callback function. Refer to the `ibnotify` function description later in this chapter for more details.

Driver Version Requirements

If you use Windows NT, you must have version 1.2 or later of the NI-488.2M driver to use `ibInstallCallback` and `ibnotify`.

If you use Windows 95, you must have version 1.1 or later of the NI-488.2M driver to use `ibInstallCallback` and `ibnotify`.

If you use Windows 3.1, you can use the limited version of `ibInstallCallback`, but you cannot use `ibnotify`.

GPIB Function Reference

The software reference manual you receive with your GPIB interface describes most of the functions in the GPIB/GPIB-488.2 Library. This section contains descriptions, in alphabetical order, only for the Device Manager functions, the callback installation functions, and the functions for returning the thread-specific status variables.

☞ **Note** `ResetDevs` ***is not available in LabWindows/CVI. This function was available in a previous LabWindows version.***

CloseDev

```
int result = CloseDev (int Device);
```

Purpose

Closes a device.

Parameter

Input

Name	Type	Description
Device	integer	The device to close.

Return Value

Name	Type	Description
result	integer	Result of the close device operation.

Return Codes

Code	Description
`-1`	Error—cannot find device.
`0`	Success.

Using This Function

Takes a device offline. `CloseDev` first calls `ibloc` and then calls `ibonl` with a value of zero. **Device** is the device descriptor you obtain when you open the device with `OpenDev`. If `CloseDev` cannot find the device descriptor in its table, it returns a `-1`. You should use `CloseDev` only in conjunction with `OpenDev`. Never call `CloseDev` with a device descriptor you obtain by calling `ibfind`.

CloseInstrDevs

```
int result = CloseInstrDevs (char *instrumentPrefix);
```

Purpose

Closes all devices associated with an instrument module.

Parameter

Input

Name	Type	Description
instrumentPrefix	string	Must be null-terminated.

Return Value

Name	Type	Description
result	integer	Result of the close instrument devices operation.

Return Code

Code	Description
`0`	Success.

Using This Function

instrumentPrefix specifies the prefix of the instrument module to close. `CloseInstrDevs` always returns zero. You should use `CloseInstrDevs` only in conjunction with `OpenDev`.

ibInstallCallback

```
int status = ibInstallCallback (int boardOrDevice, int eventMask,
                                GPIBCallbackPtr callbackFunction,
                                void *callbackData)
```

Note `ibInstallCallback` ***is available only under Windows. Under UNIX, use*** `ibsgnl`***. Under Windows 3.1, the data type of the return value and the first two parameters is*** `short` ***rather than*** `int`***.*** `ibInstallCallback` ***does not work with GPIB-ENET.***

Purpose

Allows you to install a synchronous callback function for a specified interface or device. If you want to install an asynchronous callback, use `ibnotify` instead.

The callback function is called when any of the GPIB events you specify in **eventMask** have occurred on the interface or device, but only while you allow the system to process events. The system can process events when you call `ProcessSystemEvents` or `GetUserEvent`, or when you call `RunUserInterface` and none of your callback functions are currently active. The callbacks are called “synchronous” because you can invoke them only in the context of normal event processing.

Unlike asynchronous callbacks, no restrictions exist on what you can do in a synchronous callback. On the other hand, the latency between the occurrence of a GPIB event and the invocation of the callback function is greater and more unbounded with synchronous callbacks than with asynchronous callbacks.

You can install only one callback function for each interface or device. Each call to `ibInstallCallback` for the same interface or device supersedes the previous call.

To disable callbacks for an interface or device, pass `0` for **eventMask**.

To use `ibInstallCallback` under Windows 95/NT, you must have one of the following versions of the NI-488.2M driver:

- For Windows 95: Version 1.1 or later
- For Windows NT: Version 1.2 or later

If you use Windows 3.1, you must pass a board index for the first parameter, and you can use only SRQI or CMPL for **eventMask**.

Parameters

Input

Name	Type	Description
boardOrDevice	integer (short integer on Windows 3.1)	Board index, or board or device descriptor you obtain from `OpenDev`, `ibfind`, or `ibdev`. (Under Windows 3.1, must be a board index).
eventMask	integer (short integer on Windows 3.1)	Specifies the events upon which the callback function is called. Pass `0` to disable callbacks. Refer to the following *Parameter Discussion*.
callbackFunction	GPIBCallbackPtr	Name of the user function to call when the specified events occur. Refer to the following *Parameter Discussion*.
callbackData	void pointer	Pointer to a user-defined, 4-byte value to pass to the callback function.

Return Value

Name	Type	Description
status	integer (short integer on Windows 3.1)	Same value as the `ibsta` status variable. Refer to your NI-488.2 or NI-488.2M user manual for a description of the values of `ibsta`.

Parameter Discussion

eventMask

You specify the conditions upon which LabWindows/CVI invokes the callback function as bits in **eventMask**. The bits correspond to the bits of the `ibsta` status word. This value reflects a sum of one or more events. If any one of the conditions occur, the callback is called.

If, when you install the callback, one of the bits you set in the mask is already TRUE, LabWindows/CVI invokes the callback immediately. For example, if you pass CMPL as the **eventMask**, and `ibwait` would currently return a status word with CMPL set, LabWindows/CVI calls the callback immediately.

If you use Windows 95/NT, the following mask bits are valid:

- At the board level, you can specify any of the status word bits that you can specify in the **waitMask** parameter to `ibwait` for a board, other than ERR. This includes SRQI, END, CMPL, TIMO, CIC, and others.
- At the device level, you can specify any of the status word bits that you can specify in the **waitMask** parameter to `ibwait` for a device, other than ERR. This includes RQS, END, CMPL, and TIMO.

If you use Windows 3.1, the only valid mask bits are SRQI or CMPL but not both.

SRQI, RQS, and Auto Serial Polling

If you want to install a callback for the SRQI (board-level) event, you must disable autopolling. You can disable autopolling with the following function call:

```
ibconfig (boardIndex, IbcAUTOPOLL, 0);
```

If you want to install a callback for the RQS (device-level) event, you must enable autopolling for the board. You can enable autopolling with the following function call:

```
ibconfig (boardIndex, IbcAUTOPOLL, 1);
```

callbackFunction

The callback function must have the following form:

```
void CallbackFunctionName (int boardOrDevice, int mask,
                           void *callbackData);
```

mask and **callbackData** are the same values you pass to `ibInstallCallback`.

If invoked because of an SRQI or RQS condition, the callback function must call `ibrsp` to read the status byte. For an SRQI (board-level) condition, calling the `ibrsp` function is necessary to cause the requesting device to turn off the SRQ line.

```
char statusByte;
ibrsp (device, &statusByte);
```

If invoked because of a completed asynchronous I/O operation that `ibrda`, `ibwrta`, or `ibcmda` started, the callback function must make the following call:

```
ibwait (boardOrDevice, TIMO | CMPL);
```

The `ibcnt` and `ibcntl` status variables are not updated until you call `ibwait`.

See Also

`ibnotify`

iblock

```
int status = iblock (int boardDevice);
```

Purpose

Blocks other processes from accessing the specified GPIB-ENET board or device. You can release the lock by calling `ibunlock` with the same board or device descriptor.

By default, a process returns an ELCK (21) error when it attempts to use a board or device locked by another process. You can cause your process to block instead of returning an error by making the following function call:

```
ibconfig (boardOrDevice, IbcBlockIfLocked, 1);
```

There is no timeout on the process that remains in a blocked state.

In general, you should use `iblock` to gain critical access to a GPIB-ENET board or device when multiple processes might be accessing it at the same time. When the GPIB-ENET is locked, the GPIB driver guarantees that subsequent calls you make to the GPIB-ENET are completed without interruption.

Parameters

Input

Name	Type	Description
boardDevice	integer	Device descriptor you obtain from `OpenDev`, `ibfind`, or `ibdev`.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

See Also

`ibunlock`

ibnotify

```
int status = ibnotify (int boardOrDevice, int eventMask,
                        GpibNotifyCallback_t callbackFunction,
                        void *callbackData);
```

Note `ibnotify` ***is available only under Windows 95/NT. Under UNIX, use*** `ibsgnl`**.** `ibnotify` ***does not work with GPIB-ENET.***

Purpose

Allows you to install an asynchronous callback function for a specified board or device. If you want to install a synchronous callback, use `ibInstallCallback` instead.

LabWindows/CVI calls the callback function when any of the GPIB events you specify in **eventMask** occur on the specified interface or device. LabWindows/CVI can call asynchronous callbacks at any time while your program is running. You do not have to allow the system to process events. Because of this, you are restricted in what you can do in the callback. Refer to the following *Restrictions on Operations in Asynchronous Callbacks* discussion.

You can install only one callback function for each interface or device. Each call to `ibnotify` for the same interface or device supersedes the previous call.

To disable callbacks for a interface or device, pass `0` for **eventMask**.

Parameters

Input

Name	Type	Description
boardOrDevice	integer	Board index, or board or device descriptor you obtain from `OpenDev`, `ibfind`, or `ibdev`.
eventMask	integer	Specifies the events upon which the callback function is called. Pass `0` to disable callbacks. Refer to the *Parameter Discussion*.
callbackFunction	`GpibNotifyCallback_t`	Name of the function LabWindows/CVI calls when the specified events occur. Refer to the *Parameter Discussion*.
callbackData	void pointer	Pointer to a user-defined, 4-byte value to pass to the callback function.

Return Value

Name	Type	Description
status	integer	Same value as the `ibsta` status variable. Refer to your NI-488.2M user manual for a description of the values of `ibsta`.

Parameter Discussion

eventMask

You specify the conditions upon which LabWindows/CVI invokes the callback function as bits in **eventMask**. The bits correspond to the bits of the `ibsta` status word. This value reflects a sum of one or more events. If any one of the conditions occur, the callback is called.

If, when you install the callback, one of the bits you set in the mask is already TRUE, LabWindows/CVI invokes the callback immediately. For example, if you pass CMPL as the **eventMask**, and `ibwait` would currently return a status word with CMPL set, LabWindows/CVI calls the callback immediately.

At the board level, you can specify any of the status word bits that you can specify in the **waitMask** parameter to `ibwait` for a board, other than ERR. This includes SRQI, END, CMPL, TIMO, CIC, and others.

At the device level, you can specify any of the status word bits that you can specify in the **waitMask** parameter to `ibwait` for a device, other than ERR. This includes RQS, END, CMPL, and TIMO.

SRQI, RQS, and Auto Serial Polling

If you want to install a callback for the SRQI (board-level) event, you must disable autopolling. You can disable autopolling with the following function call:

```
ibconfig (boardIndex, IbcAUTOPOLL, 0);
```

If you want to install a callback for the RQS (device-level) event, you must enable autopolling for the board. You can enable autopolling with the following function call:

```
ibconfig (boardIndex, IbcAUTOPOLL, 1);
```

callbackFunction

The callback function must have the following form:

```
void __stdcall CallbackFunctionName (int boardOrDevice, int sta,
                                     int err, long cntl, void *callbackData);
```

callbackData is the same **callbackData** value you pass to `ibInstallCallback`. **sta**, **err**, and **cntl** contain the information that you normally obtain using the `ibsta`, `iberr`, and `ibcntl` global variables or `ThreadIbsta`, `ThreadIberr`, and `ThreadIbcntl`. The global variables and thread status functions return undefined values within the callback function. So you must use **sta**, **err** and **cntl** instead.

The value you return from the callback function is very important. It is the event mask that is used to *rearm* the callback. Returning `0` disarms the callback; that is, it is not called again until you make another call to `ibnotify`. If you return an event mask different than the one you originally passed to `ibnotify`, `ibnotify` uses the new event mask. Normally, you want to return the same event mask that you originally passed to `ibnotify`.

If you return an invalid event mask or if there is an operating system error in rearming the callback, the callback is called with the **sta** set to ERR, **err** set to `EDVR`, and **cntl** set to `IBNOTIFY_REARM_FAILED`.

Caution ***Because the callback can be called as the result of a rearming error, you should always check the value of the* sta *parameter to make sure that one of the requested events has in fact occurred.***

If invoked because of an SRQI or RQS condition, the callback function should call `ibrsp` to read the status byte. For an SRQI (board-level) condition, calling `ibrsp` is necessary to cause the requesting device to turn off the SRQ line.

```
char statusByte;
ibrsp (device, &statusByte);
```

If invoked because of a completed asynchronous I/O operation that `ibrda`, `ibwrta`, or `ibcmda` started, the callback function must make the following call:

```
ibwait (boardOrDevice, TIMO | CMPL);
```

The `ibcnt` and `ibcntl` status variables are not updated until you call `ibwait`.

Restrictions on Operations in Asynchronous Callbacks

LabWindows/CVI can call callbacks you install with `ibnotify` at any time with respect to the rest of your program. You do not have to allow the system to process events. Because of this, you are restricted in what you can do in the callback. You can do the following:

- Call any GPIB function, except `ibnotify` or `ibInstallCallback`.
- Manipulate global variables, but only if you know that the callback has not been called at a point when the main part of your program is modifying or interrogating the same global variables.
- In a standalone executable, you can use any of the other LabWindows/CVI libraries, subject to the multithreading restrictions described in the documentation for each library.
- When running in the LabWindows/CVI development environment, you can use the other LabWindows/CVI libraries only in the following ways:
  - Call the User Interface Library `PostDeferredCall` function, which schedules a different callback function for *synchronous* invocation.
  - Call ANSI C functions such as `strcpy` and `sprintf`, which have no side effects and affect only the arguments you pass in. You cannot call `printf` or file I/O functions.
  - Call `malloc`, `calloc`, `realloc`, or `free`.

If you need to perform operations that fall outside these restrictions, do the following:

1. Perform the time-critical operations in the asynchronous callback and call `PostDeferredCall` to schedule a synchronous callback.
2. In the synchronous callback, perform the other operations.

See Also

`ibInstallCallback`

ibunlock

```
int status = ibunlock (int boardDevice);
```

Purpose

Releases a lock on a GPIB-ENET board or device. Refer to the `iblock` function for more information. In general, you should release your lock on a GPIB-ENET connection immediately after you make your critical access.

Parameters

Input

Name	Type	Description
boardDevice	integer	Device descriptor you obtain from `OpenDev`, `ibfind`, or `ibdev`.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

See Also

`ibunlock`

OpenDev

```
int bd = OpenDev (char *deviceName, char *instrumentPrefix);
```

Purpose

Opens a GPIB device.

Parameters

Input

Name	Type	Description
deviceName	string	Must be null-terminated.
instrumentPrefix	string	Must be null-terminated.

Return Value

Name	Type	Description
bd	integer	Result of the open device operation.

Return Code

Code	Description
`-1`	Device table is full, or no more devices are available.

Parameter Discussion

deviceName is a string that specifies a device name that appears in the `ibconf` device table. If **deviceName** is not `""`, `OpenDev` acts identically to `ibfind`. If **deviceName** is `""`, `OpenDev` acts identically to `ibdev` by opening the first available unopened device.

instrumentPrefix is a string that specifies the instrument prefix associated with the instrument module. The instrument prefix must be identical to the prefix you enter when creating the function tree for the instrument module. If the instrument module has no prefix or you are not using `OpenDev` in an instrument module, pass the string `""` for **instrumentPrefix**.

Using This Function

`OpenDev` attempts to find an unused device in the GPIB handler device table and open the device. If successful, `OpenDev` returns a device descriptor. Otherwise, it returns a negative number.

ThreadIbcnt

```
int threadSpecificCount = ThreadIbcnt (void);
```

Note `ThreadIbnct` ***is available only under Windows 95/NT.***

Purpose

Returns the value of the thread-specific `ibcnt` variable for the current thread.

The global variables `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained on a process-specific rather than a thread-specific basis. If you call GPIB functions in more than one thread, the values in these global variables are not always reliable.

Status variables analogous to `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained for each thread. `ThreadIbcnt` returns the value of the thread-specific `ibcnt` variable.

If you do not use multiple threads, `ThreadIbcnt` returns a value identical to the value of the `ibcnt` global variable.

Parameters

None.

Return Value

Name	Type	Description
threadSpecificCount	integer	Number of bytes actually transferred by the most recent GPIB read, write, or command operation for the current thread of execution. If an error occurred loading the GPIB DLL, `ThreadIbcnt` returns a Windows error code.

See Also

`ThreadIbsta`, `ThreadIberr`, `ThreadIbcntl`

ThreadIbcntl

```
long threadSpecificCount = ThreadIbcntl (void);
```

Note `ThreadIbcntl` ***is available only under Windows 95/NT.***

Purpose

Returns the value of the thread-specific `ibcntl` variable for the current thread.

The global variables `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained on a process-specific rather than a thread-specific basis. If you call GPIB functions in more than one thread, the values in these global variables are not always reliable.

Status variables analogous to `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained for each thread. `ThreadIbcntl` returns the value of the thread-specific `ibcntl` variable.

If you do not use multiple threads, `ThreadIbcntl` returns a value identical to the value of the `ibcntl` global variable.

Parameters

None.

Return Value

Name	Type	Description
threadSpecificCount	long integer	Number of bytes actually transferred by the most recent GPIB read, write, or command operation for the current thread of execution. If an error occurred loading the GPIB DLL, `ThreadIbcntl` returns a Windows error code.

See Also

`ThreadIbsta`, `ThreadIberr`, `ThreadIbcnt`

ThreadIberr

```
int threadSpecificError = ThreadIberr (void);
```

Note `ThreadIberr` ***is available only under Windows 95/NT.***

Purpose

Returns the value of the thread-specific `iberr` variable for the current thread.

The global variables `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained on a process-specific rather than a thread-specific basis. If you call GPIB functions in more than one thread, the values in these global variables are not always reliable.

Status variables analogous to `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained for each thread. `ThreadIberr` returns the value of the thread-specific `iberr` variable.

If you do not use multiple threads, `ThreadIberr` returns a value identical to the value of the `iberr` global variable.

Parameters

None.

Return Value

Name	Type	Description
threadSpecificError	integer	Most recent GPIB error code for the current thread of execution. The value is meaningful only when `ThreadIbsta` returns a value with the ERR bit set.

Return Codes

Defined Constant	Value	Description
`EDVR`	`0`	Operating system error. `ThreadIbcntl` returns the system-specific error code.
`ECIC`	`1`	Function requires GPIB-PC to be Controller-In-Charge (CIC).
`ENOL`	`2`	No listener on write function.
`EADR`	`3`	GPIB-PC addressed incorrectly.
`EARG`	`4`	Invalid function call argument.

Defined Constant	Value	Description
ESAC	5	GPIB-PC not system controller as required.
EABO	6	I/O operation aborted.
ENEB	7	Non-existent GPIB-PC interface.
EDMA	8	Virtual DMA device error.
EOIP	10	I/O started before previous operation completed.
ECAP	11	Unsupported feature.
EFSO	12	File system error.
EBUS	14	Command error during device call.
ESTB	15	Serial poll status byte lost.
ESRQ	16	SRQ stuck in the asserted state.
ETAB	20	Device list error during a `FindLstn` or `FindRQS` call.
ELCK	21	Address or board is locked.
ELNK	200	The GPIB library was not linked. Dummy functions were linked instead.
EDLL	201	Error loading `gpib-32.dll`. `ThreadIbcntl` returns the Windows error code.
EFNF	203	Unable to find the function in `gpib-32.dll`. `ThreadIbcntl` returns the Windows error code.
EGLB	205	Unable to find globals in `gpib-32.dll`. `ThreadIbcntl` returns the Windows error code.
ENNI	206	Not a National Instruments `gpib-32.dll`.
EMTX	207	Unable to acquire mutex for loading DLL. `ThreadIbcntl` returns the Windows error code.

Defined Constant	Value	Description
EMSG	210	Unable to register callback function with Windows.
ECTB	211	The callback table is full.

See Also

ThreadIbsta, ThreadIbcnt, ThreadIbcntl

ThreadIbsta

```
int threadSpecificStatus = ThreadIbsta (void);
```

Note `ThreadIbsta` ***is available only under Windows 95/NT.***

Purpose

Returns the value of the thread-specific `ibsta` variable for the current thread.

The global variables `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained on a process-specific rather than a thread-specific basis. If you call GPIB functions in more than one thread, the values in these global variables are not always reliable.

Status variables analogous to `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained for each thread. `ThreadIbsta` returns the value of the thread-specific `ibsta` variable.

If you do not use multiple threads, `ThreadIbsta` returns a value identical to the value of the `ibsta` global variable.

Parameters

None.

Return Value

Name	Type	Description
threadSpecificStatus	integer	Status value for the current thread of execution. The status value describes the state of the GPIB and the result of the most recent GPIB function call in the thread. Any value with the ERR bit set indicates an error. Call `ThreadIberr` for a specific error code.

Return Codes

The return value is a sum of the following bits.

Defined Constant	Hex Value	Description
`ERR`	`8000`	GPIB error.
`END`	`2000`	END or EOS detected.
`SRQI`	`1000`	SRQ is on.

Defined Constant	Hex Value	Description
RQS	800	Device requesting service.
CMPL	100	I/O completed.
LOK	80	GPIB-PC in lockout state.
REM	40	GPIB-PC in remote state.
CIC	20	GPIB-PC is Controller-In-Charge.
ATN	10	Attention is asserted.
TACS	8	GPIB-PC is talker.
LACS	4	GPIB-PC is listener.
DTAS	2	GPIB-PC in device trigger state.
DCAS	1	GPIB-PC in device clear state.

See Also

ThreadIberr, ThreadIbcnt, ThreadIbcntl

5

RS-232 Library

This chapter describes the functions in the LabWindows/CVI RS-232 Library. The *RS-232 Library Function Overview* section contains general information about the RS-232 Library functions and panels. The *RS-232 Library Function Reference* section contains an alphabetical list of function descriptions.

In order to use the RS-232 Library under UNIX, your UNIX kernel must support asynchronous I/O functions, for example, `aioread` and `aiowrite`. You can enable this by building your UNIX kernel as `Generic` instead of `Generic Small`.

RS-232 Library Function Overview

This section contains general information about the RS-232 Library functions and panels. The RS-232 Library also can be used with a National Instruments RS-485 serial board.

RS-232 Library Function Panels

The RS-232 Library function panels are grouped in the tree structure in Table 5-1 according to the types of operations they perform.

The first- and second-level headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels. Each RS-232 function panel generates one or more RS-232 function calls.

Table 5-1. Functions in the RS-232 Library Function Tree

Class/Panel Name	Function Name
Open/Close	
Open COM and Configure	`OpenComConfig`
Close COM	`CloseCom`
Open COM—Current State	`OpenCom`
Input/Output	
Read Buffer	`ComRd`
Read Terminated Buffer	`ComRdTerm`
Read Byte	`ComRdByte`
Read To File	`ComToFile`
Write Buffer	`ComWrt`
Write Byte	`ComWrtByte`
Write From File	`ComFromFile`

Table 5-1. Functions in the RS-232 Library Function Tree (Continued)

Class/Panel Name	Function Name
XModem	
XModem Send File	`XModemSend`
XModem Receive File	`XModemReceive`
XModem Configure	`XModemConfig`
Control	
Set Timeout Limit	`SetComTime`
Set XON/XOFF Mode	`SetXMode`
Set CTS Mode	`SetCTSMode`
Flush Input Queue	`FlushInQ`
Flush Output Queue	`FlushOutQ`
Send Break Signal	`ComBreak`
Set Escape Code	`ComSetEscape`
Status	
Get COM Status	`GetComStat`
Get Input Queue Length	`GetInQLen`
Get Output Queue Length	`GetOutQLen`
Return RS-232 Error	`ReturnRS232Err`
Get Error String	`GetRS232ErrorString`
Callbacks	
Install COM Callback	`InstallComCallback`

Class Descriptions

- The Open/Close function panels open, close, and configure a COM port.
- The Input/Output function panels read from and write to a COM port.
- The XModem function panels transfer files using the XModem protocol.
- The Control function panels set the timeout limit, set communication modes, flush the I/O queues, and send the break signal.
- The Status function panels return the COM port status and the length of the I/O queues.
- The Callbacks function panel installs callback functions for COM events.

The online help with each panel contains specific information about operating each function panel.

Using RS-485

You can use all the functions in the RS-232 Library with the National Instruments RS-485 AT-Serial board. `ComSetEscape` allows you to control the transceiver mode of the board.

Reporting RS-232 Errors

The functions in the RS-232 Library return negative values when an error occurs. In addition, the global variable `rs232err` is updated after each function call to the RS-232 Library. If the function executes properly, it sets `rs232err` to zero. Otherwise, it sets `rs232err` to the same error code that it returns.

When an aynchronous write operation fails, the function sets `rs232err` to an error code unless it already contains a negative value. `ReturnRS232Error` returns the same value as `rs232err` except that it keeps track of separate error codes for each thread in your application. In a multithreaded application on Windows 95/NT, use `ReturnRS232Error` rather than `rs232err`.

`GetRS232ErrorString` translates each possible error code into a meaningful error string.

Table 5-10 at the end of this chapter lists the possible error conditions that can occur when you use the RS-232 Library functions.

XModem File Transfer Functions

With the XModem functions, you can transfer files using a data transfer protocol. The protocol uses a generally accepted technique for serial file transfers with error-checking. The XModem functions transfer packets that contain data from the files plus error-checking and synchronization information.

You do not need to understand the protocol to use the functions. To transfer a file, open the COM port, use `XModemSend` on the sender side of the transfer or `XModemReceive` on the receiver side of the transfer, and then close the COM port. The XModem functions handle all aspects of the transfer protocol.

You can treat the XModem functions as higher-level functions that perform a more precisely defined task than the functions `ComToFile` and `ComFromFile`. Use `ComToFile` and `ComFromFile` if you need finer control over the file operations. Remember that the XModem functions calculate the checksum and retransmit when they detect an error, whereas `ComToFile` and `ComFromFile` do not do so.

Troubleshooting

Establishing communication between two RS-232 devices can be difficult because of the many different possible configurations. When using this library, you must know the device requirements, such as baud rate, parity, number of data bits, and number of stop bits. These configurations must match between the two parties of communication.

If you encounter difficulty in establishing initial communication with the device, refer to an elementary RS-232 communications handbook for information about cable requirements and

general RS-232 communication. Refer also to the *RS-232 Cable Information* section later in this chapter.

You must call `OpenCom` or `OpenComConfig` to open a COM port before using any of the other functions.

If the program writes data to the output queue and then immediately closes the COM port, the data in the queue might be lost if LabWindows/CVI did not have time to send it over the port. To guarantee that all bytes are written before the port closes, monitor the length of the output queue with `GetOutQLen`. When the output queue length becomes zero, it is safe to close the port.

If `XModemReceive` fails to complete properly, verify that the input queue length is greater than or equal to the packet size. Refer to `OpenComConfig` and `XModemConfig` for more information.

If the receiver appears to lose data the sender transmits, the input queue of the receiver might be overflowing. This means that the library does not empty the input queue of the receiver as quickly as data is coming in. You can solve this problem using handshaking, provided both devices offer the same handshaking support. Refer to the *Handshaking* section of this chapter for more information.

If an XModem file transfer with a large packet size and a low baud rate fails, you might need to increase the wait period. Ten seconds is sufficient for most transfers.

RS-232 Cable Information

An RS-232 cable consists of wires, or lines, that join two connectors. The connectors plug into the serial ports of each device to form a communications link over which data and control signals flow. Each serial port consists of numbered pins that have the meanings shown in Table 5-2.

Table 5-2. PC Cable Configuration

Pin	Meaning
2	TxD—Transmit Data *
3	RxD—Receive Data
4	RTS—Request to Send *
5	CTS—Clear to Send
6	DSR—Data Set Ready

Table 5-2. PC Cable Configuration (Continued)

Pin	Meaning
20	DTR—Data Terminal Ready *
7	Common

The items with an asterisk (*) indicate the lines that the PC drives. All other items indicate the lines the PC monitors.

The type of all serial devices is either Data Communication Equipment (DCE) or Data Transmission Equipment (DTE). IBM-compatible PCs are DTE type devices. The difference between the two types is in the meaning assigned to the pins. A DCE device reverses the meaning of pins 2 and 3, 4 and 5, and 6 and 20. In the simplest scenario, a DTE device attaches to a DCE device, such as a modem. Table 5-3 shows the cable required to connect a PC (or DTE) to a DCE device.

Table 5-3. DTE to DCE Cable Configuration

(PC)	Connect pins as indicated:	(Device)
TxD*	2——————2	RxD
RxD	3——————3	TxD*
RTS*	4——————4	CTS
CTS	5——————5	RTS*
DSR	6——————6	DTR
DTR*	20——————20	DSR*
common	7——————7	common

You need a different cable for the PC to talk to a DTE device because both devices transmit data over pin 2. A *null modem cable* connects a PC to a DTE. Table 5-4 shows the configuration of a null modem cable.

Table 5-4. PC to DTE Cable Configuration

(PC)	Connect pins as indicated:	(Device)
TxD*	2——————3	RxD
RxD	3——————2	TxD*

Table 5-4. PC to DTE Cable Configuration (Continued)

(PC)	Connect pins as indicated:	(Device)
RTS*	4——————5	CTS
CTS	5——————4	RTS*
DSR	6——————20	DTR
DTR*	20——————6	DSR*
common	7——————7	common

For more information on the meaning of DTE and DCE, refer to a reference book on RS-232 communication.

In the simplest case, a serial cable needs lines 2, 3, and 7 for basic communication to take place. Hardware handshaking and modem control can require other lines, depending on your application. Refer to the *Hardware Handshaking* section later in this chapter for more information about using the lines 4, 5, 6, and 20.

Another area that requires special attention is the *gender* of the connectors of your serial cable. The serial cable plugs into sockets in the PC and the serial device, just as a lamp cord plugs into a wall socket. Both the connector and the socket can be male, with pins (like a lamp plug), or female, with holes (like an outlet). If your serial cable connector and PC socket are the same gender, you cannot plug the cable into the socket. You can change this by attaching a small device called a *gender changer* to your cable. One type of gender changer converts a female connector to a male connector, and the other type converts a male connector to a female connector.

The size of the connector on your serial cable also can differ from the size of the socket. Most serial ports require a 25-pin connector. However, some serial ports require a 9-pin connector. To resolve this incompatibility, you must either change the connector on your serial cable or attach a small device that converts from a 25-pin connector to a 9-pin connector.

Handshaking

A common error condition in RS-232 communications is that the receiver appears to lose data the sender transmits. This condition typically results because the receiver cannot empty its input queue quickly enough.

Handshaking prevents overflow of the input queue that occurs when the receiver cannot empty its input queue as quickly as the sender is able to fill it. The RS-232 Library has two types of handshaking: software handshaking and hardware handshaking. You should enable

one or the other if you want to ensure that your application program synchronizes its data transfers with other serial devices that perform handshaking.

Software Handshaking

`SetXMode` enables software handshaking. You can use software handshaking to transfer ASCII data or text to or from a serial device that also uses software handshaking. The RS-232 Library performs software handshaking by sending and monitoring incoming data for special data bytes (XON and XOFF, or decimal 17 and 19). These bytes indicate whether the receiver is ready to receive data.

Do not enable software handshaking when you transmit binary data because the special XON/XOFF characters can occur as part of the data stream and the receiver can mistake them as control codes. However, you can enable hardware handshaking regardless of the type of data you transfer.

Software Handshaking requires no special cable configuration.

Hardware Handshaking

`SetCTSMode` enables hardware handshaking. For hardware handshaking to work, two conditions must exist. First, the serial devices must follow the same or similar hardware handshake protocols; they must use the same lines for the handshake and assign the same meanings to those lines. Second, the serial cable that connects the two devices must include the lines required to support the protocol. Because no single well-defined hardware handshake protocol exists, you must resolve any differences between the LabWindows/CVI hardware handshake protocol and the one your device uses.

Most serial devices primarily rely on the CTS and RTS lines to perform hardware handshaking and use the DTR line to signal its online presence to the other device. Some serial devices also use the DTR line for hardware handshaking, similar to the CTS line. `SetCTSMode` has two different modes to handle either case.

`SetCTSMode` employs the following line behaviors for each mode:

Note ***Under UNIX, changes to the DTR line have no effect on the communication port.***

`LWRS_HWHANDSHAKE_OFF`

The library raises the RTS and DTR lines when opening the port and lowers them when closing the port. The library sends data out the port regardless of the status of CTS.

Note ***Under Windows, you can use*** `ComSetEscape` ***to change the state of the RTS and DTR lines.***

`LWRS_HWHANDSHAKE_CTS_RTS`

- When the PC is the receiver:
  - If the port is opened, the library raises RTS and DTR.
  - If the input queue of the port is nearly full, the library lowers RTS.
  - If the input queue of the port is nearly empty, the library raises RTS.
  - If the port is closed, the library lowers RTS and DTR.
- When the PC is the sender, the RS-232 library must detect that its CTS line is high before it sends data out the port.

`LWRS_HWHANDSHAKE_CTS_RTS_DTR`

- When the PC is the receiver:
  - If the port is opened, the library raises RTS and DTR.
  - If the input queue of the port is nearly full, the library lowers RTS and DTR.
  - If the input queue of the port is nearly empty, the library raises RTS and DTR.
  - If the port is closed, the library lowers RTS and DTR.
- When the PC is the sender, the RS-232 library must detect that its CTS line is high before it sends data out the port.

Note ***The only difference between*** `LWRS_HWHANDSHAKE_CTS_RTS` ***and*** `LWRS_HWHANDSHAKE_CTS_RTS_DTR` ***is the behavior of the DTR line.***

If the handshaking mechanism of your device uses the CTS and RTS lines, refer to Table 5-3 and Table 5-4 in the previous *RS-232 Cable Information* section for information on how to configure your cable. Your cable can omit the connection between pins 6 and 20 if your device does not monitor DSR when it sends data. Notice that the RTS pin of the receiver translates to the CTS pin of the sender and that the DSR pin of the receiver translates to the DTR pin of the sender.

If you want to use hardware handshaking but your device uses a different hardware handshake protocol than the ones described here, you can build a cable that overcomes the differences. You can construct a cable to serve your special needs by referencing the pin description in Table 5-2 in the previous *RS-232 Cable Information* section.

Multithreading under Windows 95/NT

Under Windows 95/NT, you can call RS-232 Library functions from different threads in the same process, even if the functions operate on the same port. The following limitations apply:

- Do not use the `rs232err` global variable in a multithreaded application. Use `ReturnRS232Err` instead.
- Do not call `XModemReceive` or `ComToFile` from two threads at the same time if the target file in both calls is the Standard Output.

RS-232 Library Function Reference

This section describes each function in the LabWindows/CVI RS-232 Library in alphabetical order.

CloseCom

```
int result = CloseCom (int COMPort);
```

Purpose

Closes a COM port.

Parameter

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes.

Parameter Discussion

`CloseCom` does nothing if the port is not open.

ComBreak

```
int result = ComBreak (int COMPort, int breakTimeMsec);
```

Purpose

Generates a break signal.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
breakTimeMsec	integer	Range 1–255, or 0 to select 250.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes.

Using This Function

`ComBreak` generates a break signal for the number of milliseconds you indicate or for 250 ms if the **breakTimeMsec** parameter is zero. For most applications, 250 ms is adequate.

A break signal is the transmission of a special character on the communication line for a period longer than the transmission time for one character and its framing bits. You can use a break signal to convey any special condition as long as the sender and receiver agree on the meaning.

`ComBreak` returns an error if you have not opened the port or if you pass an invalid parameter value.

ComFromFile

```
int nbytes = ComFromFile (int COMPort, int fileHandle, int count,
                          int terminationByte);
```

Purpose

Reads from the specified file and writes to the output queue of a COM port.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
fileHandle	integer	File handle you obtain from `OpenFile`.
count	integer	If `0`, `ComFromFile` ignores this value.
terminationByte	integer	If `-1`, `ComFromFile` ignores this value.

Return Values

Name	Type	Description
nbytes	integer	Number of bytes written to the output queue. `<0` = error; refer to Table 5-10 for error codes

Parameter Discussion

`ComFromFile` reads **count** bytes from the file unless it encounters **terminationByte**, reaches EOF, or encounters an error. `ComFromFile` returns the number of bytes it writes successfully to the output queue. `ComFromFile` returns immediately after it places all bytes in the output queue, not when it sends all bytes out the COM port.

If **count** is zero, `ComFromFile` terminates on **terminationByte**, EOF, or error.

If **terminationByte** is `-1`, `ComFromFile` ignores it, and `ComFromFile` terminates on **count** bytes written, EOF, or error. If **terminationByte** is not `-1`, `ComFromFile` stops reading from the file when it encounters **terminationByte**. It does not write **terminationByte** to the output queue. If **terminationByte** is `CR` or `LF`, `ComFromFile` treats `CR-LF` and `LF-CR` combinations just as `ComRdTerm` does.

If **count** is `0` and **terminationByte** is `–1`, `ComFromFile` terminates on EOF or error.

Using This Function

`ComFromFile` times out whenever the library does not write any bytes from the output queue to the COM port during an entire timeout period. This can occur if you enable XON/XOFF, the device sends an XOFF character without sending the follow-on XON character, and the output queue is full. It also can occur if you enable hardware handshaking and the CTS line is not asserted. On a timeout, `ComFromFile` returns the number of bytes actually read from the COM port and sets `rs232err` to `-99`. You can set the timeout period by calling `SetComTime`.

To guarantee that `ComFromFile` removes all bytes from the output queue before it closes the port, call `GetOutQLen` to determine the number of bytes left in the output queue. If you close the port before `ComFromFile` sends every byte, you lose the bytes left in the queue.

`ComFromFile` returns an error code if you have not opened the port, if you pass an invalid parameter value, or if a file read error occurs.

ComRd

```
int nbytes = ComRd (int COMPort, char buffer[], int count);
```

Purpose

Reads **count** bytes from the input queue of the port you specify and stores them in **buffer**. Returns on timeout or when **count** bytes have been read. Returns an integer value that indicates the number of bytes read from queue.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
count	integer	0 value takes no bytes from queue.

Output

Name	Type	Description
buffer	string	Buffer in which to store the data.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes read from the input queue. `<0` = error; refer to Table 5-10 for error codes

Using This Function

`ComRd` times out whenever the input queue remains empty for an entire timeout period. On a timeout, `ComRd` returns the number of bytes actually written and sets `rs232err` to `-99`. You can set the timeout period by calling `SetComTime`.

`ComRd` returns an error code if you have not opened the port or if you pass an invalid parameter value.

Example

```
/* Read 100 bytes from input queue of COM1 into buf.*/
int n;
char buf[100];
.
.
.
n = ComRd (1, buf, 100);
if (n != 100)
   /* Timeout or error occurred before read completed. */;
```

ComRdByte

```
int byte = ComRdByte (int COMPort);
```

Purpose

Reads a byte from the input queue of the port you specify. Returns an integer whose low-order byte contains the byte read. Returns on timeout, when the byte is read, or when an error occurs. If an error or a timeout occurs, `ComRdByte` returns a negative error code. Refer to Table 5-10 at the end of this chapter for error codes. This is the only case in which the high-order byte of the return value is nonzero.

Parameter

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.

Return Value

Name	Type	Description
byte	integer	Low order byte contains the byte read. `<0` = error; refer to Table 5-10 for error codes

Using This Function

`ComRdByte` times out whenever the input queue remains empty for an entire timeout period. On a timeout, `ComRdByte` returns `-99`. You can set the timeout period by calling `SetComTime`.

`ComRdByte` returns an error code if you have not opened the port, if you pass an invalid parameter value, or if a timeout occurs.

ComRdTerm

```
int nbytes = ComRdTerm (int COMPort, char buffer[], int count,
                        int terminationByte);
```

Purpose

Reads from the input queue until **terminationByte** occurs in **buffer**, **count** is met, or a timeout occurs. Returns integer value that indicates the number of bytes read from the queue.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
count	integer	If `0`, `ComRdTerm` removes no bytes from queue.
terminationByte	integer	Low byte contains the numeric equivalent of the terminating character.

Output

Name	Type	Description
buffer	string	Buffer in which to store the data.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes read from the input queue. `<0` = error; refer to Table 5-10 for error codes

Using This Function

`ComRdTerm` times out whenever the input queue remains empty during an entire timeout period. This occurs when no data has been received during one timeout period. On a timeout, `ComRdTerm` returns the number of bytes read and sets `rs232err` to `-99`. You can set the timeout period by calling `SetComTime`.

If the read terminates on the termination byte, `ComRdTerm` neither writes the byte to the buffer nor includes it in **count**.

If the termination character is a carriage return (`CR` or decimal 13) or a linefeed (`LF` or decimal 10), the function handles it as follows:

- If **terminationByte** = `CR`, and if the character immediately following `CR` is `LF`, `ComRdTerm` discards the `LF` in addition to the `CR`.
- If **terminationByte** = `LF`, and if the character immediately following `LF` is `CR`, `ComRdTerm` discards the `CR` in addition to the `LF`.

`ComRdTerm` includes in the return count only the bytes placed in buffer. If `ComRdTerm` discards `CR` or `LF` because it follows an `LF` or `CR`, the function does not count it toward satisfying **count**.

`ComRdTerm` returns an error if you have not opened the port or you pass an invalid parameter value.

ComSetEscape

```
int result = ComSetEscape (int COMPort, int escapeCode);
```

Purpose

Directs a COM port to carry out an extended function such as clearing or setting the RTS signal line or setting the transceiver mode for RS-485. The serial device driver defines the extended functions.

Not all device drivers support all escape codes. `ComSetEscape` returns Unknown System Error (`-1`) when the device driver does not support a particular escape code.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
escapeCode	integer	Specifies the escape code of the extended function.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes.

Parameter Discussion

You can use the following values for escape code:

`CLRDTR`—Clears the DTR (data-terminal-ready) signal.

`CLRRTS`—Clears the RTS (request-to-send) signal.

`GETMAXCOM`—Returns the maximum COM port identifier the system supports. This value ranges from 0x00 to 0x7F, such that 0x00 corresponds to COM1, 0x01 to COM2, 0x02 to COM3, and so on.

`SETDTR`—Sends the DTR signal.

`SETRTS`—Sends the RTS signal.

`SETXOFF`—Causes the port to act as if it has received an XOFF character.

`SETXON`—Causes the port to act as if it has received an XON character.

You can use the following values only with the National Instruments RS-485 serial driver:

`WIRE_4`—Sets the transceiver to four-wire mode.

`WIRE_2_ECHO`—Sets the transceiver to two-wire DTR controlled with echo mode.

`WIRE_2_CTRL`—Sets the transceiver to two-wire DTR controlled without echo.

`WIRE_2_AUTO`—Sets the transceiver to two-wire auto TXRDY controlled mode.

ComToFile

```
int nbytes = ComToFile (int COMPort, int fileHandle, int count,
                        int terminationByte);
```

Purpose

Reads from the input queue of a COM port and writes the data to a file. Returns the number of bytes successfully written to the file. `ComToFile` reads bytes from the input queue until it satisfies **count**, encounters **terminationByte**, a timeout occurs, or an error occurs.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
fileHandle	integer	File handle you obtain from `OpenFile`.
count	integer	If `0`, `ComToFile` ignores this value.
terminationByte	integer	If `-1`, `ComToFile` ignores this value.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes written to the file. `<0` = error; refer to Table 5-10 for error codes

Parameter Discussion

If **count** is zero, `ComToFile` ignores it and terminates on **terminationByte** or error.

If **terminationByte** is `-1`, `ComToFile` ignores it and terminates on **count** bytes read or an error. If **terminationByte** is valid, the function stops when it encounters a byte that has the value of **terminationByte**; `ComToFile` removes the termination byte from the input queue and does not write it to the file. If **terminationByte** is `CR` or `LF`, `ComToFile` treats `CR LF` and `LF CR` combinations just as `ComRdTerm` does. If **count** is `0` and **terminationByte** is `–1`, `ComToFile` terminates on error or timeout.

Using This Function

`ComToFile` times out whenever the input queue remains empty for an entire timeout period. On a timeout, `ComToFile` returns the number of bytes actually written to the COM port and sets `rs232err` to `-99`. You can set the timeout period by calling `SetComTime`.

`ComToFile` returns an error code if you have not opened the port, if you pass an invalid parameter value, or if a file write error occurs.

ComWrt

```
int nbytes = ComWrt (int COMPort, char buffer[], int count);
```

Purpose

Writes **count** bytes to the output queue of the port you specify. Returns an integer value that indicates the number of bytes placed in the queue. Returns immediately without waiting for the bytes to be sent out of the serial port.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
buffer	string	Buffer that contains data to write or the actual string.
count	integer	`0` value places no bytes in queue.

Return Values

Name	Type	Description
nbytes	integer	Number of bytes placed in the output queue. `<0` = error; refer to Table 5-10 for error codes

Using This Function

`ComWrt` times out whenever the library does not write any bytes from the output queue to the COM port during an entire timeout period. This can occur if you enable XON/XOFF, the device sends an XOFF character without sending the follow-on XON character, and the output queue is full. It also can occur if you enable hardware handshaking and the CTS line is not asserted. On a timeout, `ComWrt` returns the number of bytes actually written and sets `rs232err` to `-99`.

`ComWrt` sends bytes from the output queue to the serial device under interrupt control without program intervention. If you close the port before all bytes are sent, you lose the bytes that remain in the queue. To guarantee that all bytes are removed from the output queue before you close the port, call `GetOutQLen`. `GetOutQLen` returns the number of bytes that remain in the output queue.

`ComWrt` returns an error if you have not opened the port or if you pass an invalid parameter value.

Example

```
/* Place the string "Hello, world!" in the output queue of */
/* COM2 and check if operation was successful. */
if (ComWrt (2, "Hello, World!", 13) != 13)
/* Operation was unsuccessful. */;
```

or

```
char buf[100];
Fmt(buf,"%s","Hello, World!");
if (ComWrt (2, buf, 13) != 13)
/* Operation was unsuccessful. */;
```

ComWrtByte

```
int status = ComWrtByte (int COMPort, int byte);
```

Purpose

Writes a byte to the output queue of a COM port. The byte written is the low-order byte of the integer. Returns a `1` to indicate the operation is successful or a negative error code to indicate the operation failed. Returns immediately without waiting for the byte to be transmitted out through the serial port.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
byte	integer	Only the low-order byte is significant.

Return Values

Name	Type	Description
status	integer	Result of the write operation. `<0` = error; refer to Table 5-10 for error codes `0` = a timeout occurred `1` = one byte placed in the output queue

Parameter Discussion

`ComWrtByte` times out whenever the library does not write any bytes from the output queue to the COM port during an entire timeout period. This can occur if you enable XON/XOFF, the device sends an XOFF character without sending the follow-on XON character, and the output queue is full. It also can occur if you enable hardware handshaking and the CTS line is not asserted. On a timeout, `ComWrtByte` returns `0` and sets `rs232err` to `-99`.

`ComWrtByte` sends bytes from the output queue to the serial device under interrupt control without program intervention. If you close the port before all bytes are sent, you lose the bytes left in the queue. To guarantee that all bytes are removed from the output queue before you close the port, call `GetOutQLen`. `GetOutQLen` returns the number of bytes left in the output queue.

`ComWrtByte` returns an error if you have not opened the port or if you pass an invalid parameter value.

FlushInQ

```
int status = FlushInQ (int COMPort);
```

Purpose

Removes all characters from the input queue of a COM port.

Parameter

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.

Return Value

Name	Type	Description
status	integer	Refer to Table 5-10 for error codes.

Using This Function

You can use `FlushInQ` to flush a flawed transmission in preparation for re-transmission. It alleviates the need to read bytes into a buffer to empty the queue. If the queue is already empty, `FlushInQ` does nothing.

`FlushInQ` returns a negative error code if you have not opened the port of if you pass an invalid value for **COMPort**.

FlushOutQ

```
int status = FlushOutQ (int COMPort);
```

Purpose

Removes all characters from the output queue of a COM port.

Parameter

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.

Return Value

Name	Type	Description
status	integer	Refer to Table 5-10 for error codes.

Using This Function

`FlushOutQ` returns an error if you have not opened the port of if you pass an invalid value for **COMPort**.

GetComStat

```
int status = GetComStat (int COMPort);
```

Purpose

Returns information about the status of a COM port. The library accumulates COM port conditions until you call `GetComStat`.

Parameter

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.

Return Values

Name	Type	Description
status	integer	Bits indicate COM port status. `<0` = error; refer to Table 5-10 for error codes

Using This Function

Table 5-5 lists definitions of specific bits in the return value. Several bits can indicate the presence of more than one condition.

Table 5-5. Bit Definitions for the GetComStat Function

Hex Value	Mnemonic	Description
0001	`INPUT LOST`	Input queue filled and input characters lost; you did not remove characters fast enough.
0002	`ASYNCH ERROR`	Problem determining number of characters in input queue. This is an internal error and normally should not occur.
0010	`PARITY`	Parity error detected.
0020	`OVERRUN`	Overrun error detected; a character was received before the receiver data register was emptied.
0040	`FRAMING`	Framing error detected; stop bits were not received when expected.

Table 5-5. Bit Definitions for the GetComStat Function (Continued)

Hex Value	Mnemonic	Description
0080	`BREAK`	Break signal detected.
1000	`REMOTE XOFF`	XOFF character received. If you enabled XON/XOFF, no characters are removed from the output queue and sent to the other device until that device sends an XON character. Refer to `SetXMode`.
4000	`LOCAL XOFF`	XOFF character sent to the other device. If you enabled XON/XOFF, XOFF is transmitted when the input queue is 50%, 75%, and 90% full. If the other device is sensitive to XON/XOFF protocol, it transmits no more characters until it receives an XON character. You use this process to avoid the INPUT LOST error.

Notice the ambiguity in this status information. If an error occurs on the indicated port, your application program knows that one or more bytes are invalid. Your program cannot determine from the status word which byte read is invalid.

`GetComStat` returns a negative error code if you have not opened the port of if you pass an invalid value for **COMPort**.

GetInQLen

```
int len = GetInQLen (int COMPort);
```

Purpose

Returns the number of characters in the input queue of a COM port. `GetInQLen` does not change the input queue.

Parameter

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.

Return Value

Name	Type	Description
len	integer	Number of characters in the input queue.

Parameter Discussion

`GetInQLen` returns an error if you have not opened the port of if you pass an invalid value for **COMPort**.

GetOutQLen

```
int len = GetOutQLen (int COMPort);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `GetOutQLen`**.**

Purpose

Returns the number of characters in the output queue of a COM port.

Parameter

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.

Return Value

Name	Type	Description
len	integer	Number of characters in the output queue.

Using This Function

You can use `GetOutQLen` to ensure the output queue empties before you close the port. This function has no effect on the output queue.

`GetOutQLen` returns an error if you have not opened the port of if you pass an invalid value for **COMPort**.

GetRS232ErrorString

```
char *message = GetRS232ErrorString (int errorNum);
```

Purpose

Converts the error number an RS-232 Library function returns into a meaningful error message.

If **errorNum** is `-1` (Unknown System Error) and you are running under Windows 95/NT, `GetRS232ErrorString` calls the Windows SDK `GetLastError` function and translates the return value to a Windows message string.

Parameter

Input

Name	Type	Description
errorNum	integer	Error code an RS-232 function returns.

Return Value

Name	Type	Description
message	string	Explanation of error.

InstallComCallback

```
int status = InstallComCallback (int COMPort, int eventMask, int notifyCount,
                                 int eventCharacter, ComCallbackPtr callbackPtr,
                                 void *callbackData);
```

☞ **Note** ***Only the Windows version of LabWindows/CVI supports*** `InstallComCallback`**.**

Purpose

Allows you to install a callback function for a particular COM port. The callback function is called whenever any of the events you specify in **eventMask** occur on the COM port and you allow the system to process events. The system can process events in the following situations:

- You call `RunUserInterface` and none of your callback functions is currently executing.
- You call `GetUserEvent`.
- You call `ProcessSystemEvents`.

You can install only one callback function for each COM port. Each call to this function for the same COM port supersedes the previous call.

To disable callbacks for a board or device, pass `0` for the **eventMask** and/or **callbackFunction**.

☞ **Note** ***The callback function can receive more than one event at a time. When using this function at higher baud rates, the library might miss some*** `LWRS_RXCHAR` ***events. Use*** `LWRS_RECEIVE` ***or*** `LWRS_RXFLAG` ***instead.***

☞ **Note** ***Once the*** `LWRS_RECEIVE` ***event occurs, it does not occur again until the input queue falls below and then rises back above*** **notifyCount** ***bytes.***

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
eventMask	integer	Events upon which the callback function is called. Refer to the following *Parameter Discussion* for a list of valid events. If you want to disable callbacks, pass `0`.
notifyCount	integer	Minimum number of bytes the input queue must contain before it sends the `LWRS_RECEIVE` event to the callback function. Valid range = 0 to size of input queue
eventCharacter	integer	Specifies the character or byte value that triggers the `LWRS_RXFLAG` event. Valid range = 0 to 255
callbackPtr	ComCallbackPtr	Name of the user function that processes the event callback.
callbackData	void *	A 4-byte value the library passes to the callback function.

Return Value

Name	Type	Description
status	integer	Refer to Table 5-10 for error codes.

Parameter Discussion

The callback function must have the following form:

```
void CallbackFunctionName (int COMPort, int eventMask,
                           void *callbackData);
```

eventMask and **callbackData** are the same values you pass to `InstallComCallback`. You can use **callbackData** as a pointer to a data object you want to access in the callback function. In this way, you do not have to declare the data object as a global variable.

You specify the events using bits in **eventMask**, and you can specify multiple event bits. Table 5-6 lists valid event bits.

Table 5-6. Valid Event Bits and Descriptions

Bit	Hex Value	COM Port Event	Constant Name	Description
0	0x0001	Any character received.	`LWRS_RXCHAR`	Set when a character is received and placed in the input queue.
1	0x0002	Received certain character.	`LWRS_RXFLAG`	Set when the event character is received and placed in the input queue. The event character is specified in **eventCharacter**.
2	0x0004	Transmit queue empty.	`LWRS_TXEMPTY`	Set when the last character in the output queue is sent.
3	0x0008	CTS changed state.	`LWRS_CTS`	Set when the CTS (clear-to-send) line changes state.
4	0x0010	DSR changed state.	`LWRS_DSR`	Set when the DSR (data-set-ready) line changes state.
5	0x0020	RLSD changed state.	`LWRS_RLSD`	Set when the RLSD (receive-line-signal-detect) line changes state.
6	0x0040	BREAK received.	`LWRS_BREAK`	Set when a break is detected on input.
7	0x0080	Line status error occurred.	`LWRS_ERR`	Set when a line-status error occurs. Line-status errors are `CE_FRAME`, `CE_OVERRUN`, and `CE_RXPARITY`.

Table 5-6. Valid Event Bits and Descriptions (Continued)

Bit	Hex Value	COM Port Event	Constant Name	Description
8	0x0100	Ring signal detected.	LWRS_RING	Set to indicate that a ring indicator was detected.
15	0x8000	**notifyCount** bytes in input queue.	LWRS_RECEIVE	Set to detect when at least **notifyCount** bytes are in the input queue. Once this event has occurred, it does not trigger again until the input queue falls below and then rises back above **notifyCount** bytes.

Example

```
notifyCount = 50; /* Wait for at least 50 bytes in queue. */
eventChar   = 13; /* Wait for LF. */
eventMask   = LWRS_RXFLAG | LWRS_TXEMPTY | LWRS_RECEIVE;
InstallComCallback (comport, eventMask, notifyCount,
                    eventChar, ComCallback, NULL);
.
.
.
/* Callback Function */
void ComCallback(int portNo, int evnetMask, void *data)
{
   if (eventMask & LWRS_RXFLAG)
      printf("Received specified character\n");
   if (eventMask & LWRS_TXEMPTY)
      printf("Transmit queue now empty\n");
   if (eventMask & LWRS_RECEIVE)
      printf("50 or more bytes in input queue\n");
}
```

OpenCom

```
int result = OpenCom (int COMPort, char deviceName[]);
```

Purpose

Opens a COM port using the default settings for the port parameters. If you want to set port settings, call `OpenComConfig` instead.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
deviceName	string	Name of the COM port.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes.

Parameter Discussion

deviceName is the name of the COM port in ASCII string format; for example, `"COM1"` for COM port 1 under Windows using `com.drv` and `"/dev/ttya"` for COM port 1 under UNIX using the Zilog 8530 SCC serial comm driver.

If you pass a `NULL` pointer or an empty string for **deviceName**, the library uses the following device names depending on the COM port number you specify. Table 5-7 shows the syntax for opening ports one through four. You can follow this model exemplified in the table to open higher-numbered ports.

Table 5-7. Syntax for Opening Ports

Port Number	deviceName under Windows	deviceName under UNIX
1	`"COM1"`	`"/dev/ttya"`
2	`"COM2"`	`"/dev/ttyb"`
3	`"COM3"`	`"/dev/ttys1"`
4	`"COM4"`	`"/dev/ttys2"`

Using This Function

`OpenCom` uses 512 bytes of the buffer for the input queue, and 512 bytes for the output. `OpenCom` assumes the default baud rate, parity, stop bits, data bits, port address, and handshake mode established through the *COM port* configuration of the operating system. The timeout for I/O operations is 5 seconds. Refer to `SetXMode`, `SetCTSMode`, `SetComTime`, and `OpenComConfig` if you want to change these defaults.

If the specified port is already open, `OpenCom` closes the port and then opens it again.

OpenComConfig

```
int result = OpenComConfig (int COMPort, char deviceName[], long baudRate,
                            int parity, int dataBits, int stopBits,
                            int inputQueueSize, int outputQueueSize);
```

Purpose

Opens a COM port and sets port parameters. If **inputQueueSize** or **outputQueueSize** is between 1 and 29, `OpenComConfig` forces it to 30.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
deviceName	string	Name of the COM port.
baudRate	long	110, 150, 300, 600, 1,200, 2,400, 4,800, 9,600, 14,400, 19,200, 28,800, 38,400, 56,000, 57,600, 115,200, 128,000, or 256,000 SPARCstations do not support 14,400, 28,800, 56,000, 57,600, 115,200, 128,000, and 256,000. PCs do not support 150. Some PC serial drivers do not support 115,200, 128,000, and 256,000.
parity	integer	`0` = no parity `1` = odd parity `2` = even parity `3` = mark parity `4` = space parity
dataBits	integer	5, 6, 7, or 8
stopBits	integer	1 or 2
inputQueueSize	integer	0 selects 512. Refer to the following *Parameter Discussion*.
outputQueueSize	integer	0 selects 512. Refer to the following *Parameter Discussion*.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes.

Parameter Discussion

deviceName is the name of the COM port in ASCII string format; for example, `"COM1"` for COM port 1 under Windows using `comm.drv` and `"/dev/ttya"` for COM port 1 under UNIX using the Zilog 8530 SCC serial comm driver.

If you pass a `NULL` pointer or an empty string for **deviceName**, the library uses the following device names depending on the COM port number you specify. Table 5-8 shows the syntax for opening ports one through four. You can follow this model to open higher-numbered ports.

Table 5-8. Syntax for Opening Ports

Port Number	deviceName on Windows	deviceName on UNIX
1	`"COM1"`	`"/dev/ttya"`
2	`"COM2"`	`"/dev/ttyb"`
3	`"COM3"`	`"/dev/ttys1"`
4	`"COM4"`	`"/dev/ttys2"`

Under UNIX, `OpenComConfig` ignores **inputQueueSize** and **outputQueueSize**. The serial driver determines the queue size.

Under Windows, if you specify `0` for **inputQueueSize** or **outputQueueSize**, `OpenComConfig` uses 512. If you specify a value between 0 and 30, `OpenComConfig` uses 30. Under Windows 95/NT, there is no maximum limitation on the queue size. Under Windows 3.1, the maximum queue size is 65,535. However, some serial drivers have a maximum of 32,767 and give undefined behavior when you use a larger queue size. National Instruments recommends that you use a queue size no greater than 32,767.

If you pass an odd number for **inputQueueSize** under Windows NT, LabWindows/CVI rounds it to the next highest even number. The Microsoft serial driver for Windows NT requires an even number for the input queue size.

On all Windows platforms, LabWindows/CVI passes **inputQueueSize** to the Windows serial driver, which might use a different number. For example, the Windows NT serial driver imposes a minimum input queue size of 4096.

Under Windows 3.1, the **baudRate** value can range from `0` to `0xffff`. The COM driver interprets values below `0xff00` literally. Values from `0xff00` to `0xffff` are codes the particular COM driver defines to represent rates higher than `0xfeff`.

Under Windows 95/NT, the comm driver interprets all **baudRate** values literally.

Using This Function

`OpenComConfig` disables XON/XOFF mode and CTS hardware handshaking. The default timeout for I/O operations is 5 seconds. Refer to the `SetXMode`, `SetCTSMode`, and `SetComTime` function descriptions if you want to change these defaults.

If the specified port is already open, `OpenComConfig` closes the port and then opens it again. Refer to the `CloseCom` function description for more information.

ReturnRS232Err

```
int status = ReturnRS232Err (void);
```

Purpose

Returns the error code from the most recent function call in the current thread. If the most recent function call was successful, `ReturnRS232Err` returns zero.

Note ***If the most recent function call was successful but an asyncrhonous write operation failed,*** `ReturnRS232Err` ***returns the error code from the asynchronous operation. If you want to make sure that a particular function call succeeded, use the return value from that function call.***

`ReturnRS232Err` is multithread safe. Using the `rs232err` global variable is not multithread safe.

If this function returns `-1` and you are running under Windows 95/NT, you can call `GetRS232ErrorString` to obtain a Windows system message.

Parameters

None.

Return Value

Name	Type	Description
status	integer	Refer to Table 5-10 for error codes.

SetComTime

```
int result = SetComTime (int COMPort, double timeoutSeconds);
```

Purpose

Sets timeout limit for input/output operations.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
timeoutSeconds	double-precision	Timeout period for all read/write functions.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes.

Using This Function

`SetComTime` sets the timeout parameters for all read and write operations. The default value of **timeoutSeconds** is 5 seconds.

For an RS-232 read operation, **timeoutSeconds** specifies the time the library allows from the start of the transfer to the arrival of the first byte. It also specifies the time the library allows to elapse between the arrival of any two consecutive bytes. An RS-232 read operation waits for at least the amount of time you specify for the next incoming byte before it returns a timeout error.

For an RS-232 write operation, **timeoutSeconds** specifies the time the library allows before the first byte is transferred to the output queue. It also specifies the time the library allows between the transfer of any two consecutive bytes to the output queue. The transfer of bytes to the output queue can stall if the output queue is full and hardware or software handshaking is held off. If the holdoff is not resolved within the timeout period, the RS-232 write operation returns a timeout error.

If **timeoutSeconds** is zero, it disables timeouts, and the read or write functions wait indefinitely for the operation to complete.

`SetComTime` returns an error if you have not opened the port or if you pass an invalid parameter value.

SetCTSMode

```
int result = SetCTSMode (int COMPort, int mode);
```

Purpose

Enables or disables hardware handshaking as described in the *Hardware Handshaking* section of the *RS-232 Library Function Overview* section at the beginning of this chapter.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
mode	integer	`0` to disable hardware handshaking; nonzero to enable hardware handshaking. Refer to the following *Parameter Discussion*.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes.

Parameter Discussion

Table 5-9 shows the valid values for **mode**.

Table 5-9. Valid Mode Values

Value	Defined Constant	Description
0	`LWRS_HWHANDSHAKE_OFF`	Hardware handshaking is disabled. The library ignores the CTS line, and raises the RTS and DTR lines the entire time the port is open.
1	`LWRS_HWHANDSHAKE_CTS_RTS_DTR`	Hardware handshaking is enabled. The library monitors the CTS line and uses the RTS and DTR lines for handshaking.
2	`LWRS_HWHANDSHAKE_CTS_RTS`	Hardware handshaking is enabled. The library monitors the CTS line, uses the RTS for handshaking, and raises the DTR line the entire time the port is open.

Using This Function

By default, hardware handshaking is not used.

`SetCTSMode` returns an error if you have not opened the port or if you pass an invalid parameter value.

SetXMode

```
int result = SetXMode (int COMPort, int mode);
```

Purpose

Enables or disables software handshaking by enabling or disabling XON/XOFF sensitivity on transmission and reception of data.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
mode	integer	`0` to disable; nonzero to enable.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes.

Using This Function

By default, XON/XOFF sensitivity is disabled. Refer to the *Software Handshaking* section of the *RS-232 Library Function Overview* section at the beginning of this chapter for more information.

`SetXMode` returns an error if you have not opened the port of if you pass an invalid parameter value.

XModemConfig

```
int result = XModemConfig (int COMPort, double startDelay,
                           int maximum#ofRetries, double waitPeriod,
                           int packetSize);
```

Purpose

Sets the XModem configuration parameters for a COM port.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
startDelay	double-precision	`0.0` selects the default value 10.0 seconds.
maximum#ofRetries	integer	`0` selects the default value 10.
waitPeriod	double-precision	`0.0` selects the default value 10.0 seconds. National Instruments recommends `>5.0`.
packetSize	integer	`0` selects the default value 128.

Return Values

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes. `0` = success

Parameter Discussion

`XModemSend` and `XModemReceive` use the baud rate and the input/output queue sizes you specify when you call `OpenComConfig`. They ignore the data bits, the parity, and the stop bits settings of `OpenComConfig` and always use 8 bits, no parity, and one stop bit. Instead of using the timeout value you set by calling `SetComTime`, XModem functions use a 1-second timeout between data bytes.

A zero input for any parameter except **COMPort** sets the parameter to its default value.

startDelay sets the timing for the initial connection between the two communication parties. When a LabWindows/CVI program assumes the role of receiver, **startDelay** specifies the interval, in seconds, during which to send the initial negative acknowledgment character to the transmitter. `XModemConfig` sends that character every **startDelay** seconds, up to **maximum#ofRetries** times. When a LabWindows/CVI program assumes the role of

transmitter, **startDelay** specifies the interval, in seconds, during which the transmitter waits for the initial negative acknowledgment. The transmitter waits up to **startDelay** × **maximum#ofRetries** seconds. The default value of **startDelay** is 10.0.

maximum#ofRetries sets the maximum number of times the transmitter retries sending a packet to the receiver on the occurrence of an error condition. The default value of **maximum#ofRetries** is 10.

waitPeriod sets the period of time between the transfers of two packets. When a LabWindows/CVI program assumes the role of transmitter, it waits for up to **waitPeriod** seconds for an acknowledgment before it re-sends the current packet. When LabWindows/CVI plays the role of receiver, it waits for up to **waitPeriod** seconds for the next packet after it sends out an acknowledgment for the current packet. If it does not receive the next packet within **delayPeriod** seconds, it re-sends the acknowledgment and waits again, up to **maximum#ofRetries** times. The default value of **waitPeriod** is 10.0.

packetSize sets the packet size in bytes. The value must be less than or equal to the input and output queue sizes. The standard XModem protocol defines packet sizes as 128 or 1,024. If you use any other size, make sure the two communication parties understand each other. The default value of **packetSize** is 128.

Using This Function

For transfers with a large packet size and a low baud rate, a large delay period is recommended.

XModemReceive

```
int result = XModemReceive (int COMPort, char fileName[]);
```

Purpose

Receives packets of information over a COM port and writes the packets to a file.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
fileName	string	Contains the pathname.

Return Value

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes. `0` = success

Using This Function

`XModemReceive` uses the XModem file transfer protocol. The transmitter also must follow this protocol for this function to work properly.

The Xmodem protocol requires that the sender and receiver agree on the error checking protocol. The sender and receiver negotiate this agreement at the beginning of the transfer, which can cause a significant delay. **XModemReceive** tries (**maximum#ofTries** + 1)/2 times to negotiate a CRC error check transfer. If there is no response, it tries to negotiate a check sum transfer up to (**maximum#ofTries** – 1)/2 times.

`XModemReceive` opens the file in binary mode and does not treat carriage returns and linefeeds as special characters. The function writes them to the RS-232 line untouched.

If the size of the file being sent is not an even multiple of the packet size, the file received is padded with ASCII NUL (0) bytes. For example, if the file being sent contains only the string `HELLO`, the file written to disk contains the letters `HELLO` followed by (packet size – 5) NUL bytes. If the packet size is 128, the file contains the five letters in `HELLO` and 123 NUL bytes.

The standard XModem protocol supports only 128 and 1,024 as packet sizes. The sender sends an SOH ASCII character (0x01) to indicate that the packet size is 128 or an STX ASCII character (0x02) to indicate that the packet size is 1,024. LabWindows/CVI attempts to support any packet size. As a receiver, when LabWindows/CVI receives an STX character from the sender, it switches to 1,024 packet size regardless of what you specify. When it receives an SOH character from the sender, it uses the packet size you specify. You can specify the packet size by calling `XModemConfig`.

For transfers with a large packet size and a low baud rate, National Instruments recommends a large delay period.

Example

```
/* Receive the file c:\test\data from COM1. */
/* NOTE:  use \\ in pathname in C instead of \. */
int n;
OpenComConfig(1, "", 9600, 1, 8, 1, 256, 256);
n = XModemReceive (1, "c:\\test\\data");
if (n != 0)
   FmtOut ("Error %d in receiving file",rs232err);
else
   FmtOut ("File successfully received.");
```

XModemSend

```
int result = XModemSend (int COMPort, char fileName[]);
```

Purpose

Reads data from a file and sends it in packets over a COM port.

Parameters

Input

Name	Type	Description
COMPort	integer	Range 1–1,000.
fileName	string	Contains the pathname.

Return Values

Name	Type	Description
result	integer	Refer to Table 5-10 for error codes. `0` = success

Using This Function

`XModemSend` opens the file in binary mode and does not treat carriage returns and linefeeds as special characters. The function sends them to the receiver untouched.

`XModemSend` uses the XModem file transfer protocol. The receiver also must follow this protocol for this function to work properly.

If the size of the file being sent is not an even multiple of the packet size, the last packet is padded with ASCII NUL (0) bytes. For example, if the file being sent contains only the string `HELLO` and the packet size is 128, the packet of data sent contains the letters `HELLO` followed by 123 (packet size – 5) NUL bytes.

The standard XModem protocol supports only 128 and 1,024 as packet sizes. The sender sends an SOH ASCII character (0x01) to indicate that the packet size is 128 or an STX ASCII character (0x02) to indicate that the packet size is 1,024. LabWindows/CVI attempts to support any packet size. As a sender, LabWindows/CVI sends an STX character when you specify packet size as 1,024. For any other packet size, it sends an SOH character. You can specify the packet size by calling `XModemConfig`.

For transfers with a large packet size and a low baud rate, National Instruments recommends a large delay period.

Error Conditions

If an error condition occurs during a call to any of the functions in the LabWindows/CVI RS-232 Library, the function returns an error code. This code is a negative value that specifies the type of error that occurred.

You can call `ReturnRS232Err` to obtain the error code, either zero or negative, from the most recent function call in the current thread. If the most recent call was successful but an asynchronous write operation failed, `ReturnRS232Err` returns the error code from the asynchronous operation. To make sure that a particular function call succeeded, use the return value from that function call.

The `rs232err` global variable is the error code from the most recent function call or failed asynchronous write operation in your application, regardless of thread. In multithreaded applications, use `ReturnRS232Err` rather than `rs232err`.

Table 5-10 lists the currently defined error codes and their meanings.

Table 5-10. RS-232 Library Error Codes

Code	Error Message
-1	Unknown system error. Refer to the following discussion.
-2	Invalid port number.
-3	Port is not open.
-4	Unknown I/O error.
-5	Unexpected internal error.
-6	No serial port found.
-7	Cannot open port.
-11	Memory allocation error in creating buffers.
-12	Unable to allocate system resources.
-13	Invalid parameter.
-14	Invalid baud rate.
-24	Invalid parity.
-34	Illegal number of data bits.
-44	Illegal number of stop bits.

Table 5-10. RS-232 Library Error Codes (Continued)

Code	Error Message
-90	Bad file handle.
-91	Error in performing file I/O.
-94	Invalid count; must be greater than or equal to 0.
-97	Invalid interrupt level.
-99	I/O operation timed out.
-104	Value must be between 0 and 255.
-114	Requested input queue size must be 0 or greater.
-124	Requested output queue size must be 0 or greater.
-151	General I/O error.
-152	Buffer parameter is NULL.
-257	Packet was sent, but no acknowledgment was received.
-258	Packet not sent within retry limit.
-259	Packet not received within retry limit.
-260	End of transmission character encountered when start of data character expected.
-261	Packet number could not be read.
-262	Packet number inconsistency.
-263	Packet data could not be read.
-264	Checksum could not be read.
-265	Checksum received did not match computed checksum.
-269	Packet size exceeds input queue size.
-300	Error opening file.
-301	Error reading file.
-302	Did not receive the initial negative acknowledgment character.
-303	Did not receive acknowledgment after the end of transmission character was sent.

Table 5-10. RS-232 Library Error Codes (Continued)

Code	Error Message
-304	Error while writing to file.
-305	Did not receive either a start of data or end of transmission character when expected.
-402	Transfer was canceled because the CAN ASCII character was received.
-503	Invalid start delay.
-504	Invalid maximum number of retries.
-505	Invalid wait period.
-506	Invalid packet size.
-507	Unable to read Cyclical Redundancy Check.
-508	Cyclical Redundancy Check error.

Errors above 200 occur only on `XModem` function calls. The library records errors `261` through `265` when the maximum number of retries has been exhausted in trying to receive an `XModem` function packet.

For error code `-1` (UnknownSystemError) on Windows 95/NT, call the `GetRS232ErrorString` function to obtain a specific Window message string.

6

DDE Library

This chapter describes the functions in the LabWindows/CVI DDE (Dynamic Data Exchange) Library. The *DDE Library Function Overview* section contains general information about the DDE Library functions and panels. The *DDE Library Function Reference* section contains an alphabetical list of function descriptions. This library is available for LabWindows/CVI for Windows only.

DDE Library Function Overview

The DDE Library includes functions specifically for Windows DDE support. This section contains general information about the DDE Library functions and panels.

DDE Library Function Panels

The DDE Library function are grouped in the tree structure in Table 6-1 according to the types of operations they perform.

The first- and second-level headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels. Each DDE function panel generates one or more DDE function calls.

Table 6-1. Functions in the DDE Library Function Tree

Class/Panel Name	Function Name
Server Functions	
Register DDE Server	`RegisterDDEServer`
Server DDE Write	`ServerDDEWrite`
Advise DDE Data Ready	`AdviseDDEDataReady`
Broadcast DDE Data Ready	`BroadcastDDEDataReady`
Unregister DDE Server	`UnregisterDDEServer`
Client Functions	
Connect to DDE Server	`ConnectToDDEServer`
Client DDE Read	`ClientDDERead`
Client DDE Write	`ClientDDEWrite`
Client DDE Execute	`ClientDDEExecute`
Set Up DDE Hot Link	`SetUpDDEHotLink`
Set Up DDE Warm Link	`SetUpDDEWarmLink`
Terminate DDE Link	`TerminateDDELink`

Table 6-1. Functions in the DDE Library Function Tree (Continued)

Class/Panel Name	Function Name
Client Functions (continued)	
Disconnect from DDE Server	`DisconnectFromDDEServer`
Get Error String	`GetDDEErrorString`

The online help with each panel contains specific information about operating each function panel.

DDE Clients and Servers

Interprocess communication with DDE involves a client and a server in each DDE conversation. A DDE server can execute commands another application sends, and send and receive information to and from a client application under Windows. A DDE client can send commands to a server application to execute and can request data from a server application.

With the LabWindows/CVI DDE Library, you can write programs that act as a DDE client or server. Refer to the *DDE Library Example Using Microsoft Excel and LabWindows/CVI* section later in this chapter for information on how to use the DDE Library functions.

To connect to a DDE server from a LabWindows/CVI program, you must know some information about the application to which you want to connect. All DDE server applications have a name and a topic that defines the connection. For example, you can connect to Microsoft Excel in two ways with the `ConnectToDDEServer` function. If you want Excel to perform tasks, such as opening worksheets and creating charts when you send commands, you should specify `excel` as the server name and `system` as the topic name in the call to the `ConnectToDDEServer` function. However, if you want to send data to an Excel spreadsheet, you should specify `excel` as the server name and the filename of the worksheet that is already loaded in Excel as the topic name.

If your program acts as a DDE server, to which other Windows applications will send and receive commands and data, you need to call `RegisterDDEServer` in your program. `RegisterDDEServer` establishes your program as a valid DDE server so that other applications can connect to it and exchange information. The server callback function is then invoked as discussed in the following section.

DDE Callback Function

Callback functions provide the mechanism for sending and receiving data to and from other applications through DDE. Similar to the method in which a callback function responds to user interface events from your User Interface Library objects, a DDE callback function responds to incoming DDE information.

A callback function in a client application can respond to only two types of DDE messages: `DDE_DISCONNECT` and `DDE_DATAREADY`. If you set up a warm link or hot link, also called an advisory loop, the callback function you specify in `ConnectToDDEServer` is called with the `DDE_DATAREADY` message whenever the data values change in the server application.

A DDE callback function used in a server application can be triggered in a number of ways. Whenever a client application attempts to connect to your server program or requests information from your program, the callback function in your program is executed to process the request. The parameter prototypes for the DDE callback functions in LabWindows/CVI are defined as follows:

```
int CallbackFunction (int handle, char *topicName, char *itemName,
                      int xType, int dataFmt, int dataSize,
                      void *dataPtr, void *callbackData);
```

Parameters

Input

Name	Type	Description
handle	integer	Conversation handle that uniquely identifies the client-server connection.
topicName	char pointer	Server application that triggers the callback.
itemName	char pointer	Data item within the server application that triggers the callback. Exception: When **xType** is `DDE_EXECUTE`, **itemName** represents the command string from the client program.
xType	integer	Transaction type; refer to Table 6-2 in the following *Transaction Types* section.
dataFmt	integer	Format of the data being transmitted.
dataSize	integer	Number of bytes in the data. Might actually be greater than the number of bytes transmitted. You should encode size information in your data.
dataPtr	void pointer	Points to the transmitted data.
callbackData	void pointer	User-defined data value.

☞ **Note** ***The value of the* dataSize *parameter is greater than or equal to the actual size of the data. You should encode more exact size information in your data.***

Return Value

The callback function should return `1` to indicate success or `0` to indicate failure or rejection of the requested action.

Transaction Types

Table 6-2 lists all the DDE transaction types, **xType**, that can trigger a callback function.

Table 6-2. DDE Transaction Types (xType)

xType	Server	Client	When?
`DDE_CONNECT`	Y	N	When a new client requests a connection.
`DDE_DISCONNECT`	Y	Y	When conversation partner quits.
`DDE_DATAREADY`	Y	Y	When conversation partner sends data.
`DDE_REQUESTDATA`	Y	N	When client requests data.
`DDE_ADVISELOOP`	Y	N	When client requests advisory loop.
`DDE_ADVISESTOP`	Y	N	When client terminates request for advisory loop.
`DDE_EXECUTE`	Y	N	When client requests execution of a command.

Refer to the description for `RegisterDDEServer` and `ConnectToDDEServer` for more information about the DDE callback function.

DDE Links

You need a DDE data link whenever a client program needs to know about changes to the value of a particular data item in the server application. You can establish a DDE data link in LabWindows/CVI by calling `SetUpDDEWarmLink` or `SetUpDDEHotLink`. Whenever the data value changes, the client callback function is invoked with the `DDE_DATAREADY` message, and the data is available in the **dataPtr** parameter.

Within one client-server connection, multiple data links can exist, each applying to a different data item. For example, you can establish a link between your LabWindows/CVI program and a particular cell in Excel. You specify the data item to which the link applies in the **itemName** parameter to `SetUpDDEWarmLink` or `SetUpDDEHotLink`.

As defined in Windows, warm and hot links differ in that under a warm link, the client is merely alerted when the data value changes, and under a hot link, the data is actually sent.

LabWindows/CVI makes no distinction between warm links and hot links. In both cases, your client application receives the data through the client callback function when the data value changes. If a warm link is in effect, LabWindows/CVI requests and receives the data from the server before the callback function is called. `SetUpDDEWarmLink` and `SetUpDDEHotLink` are provided because some DDE server applications offer only one type of link.

DDE Library Example Using Microsoft Excel and LabWindows/CVI

LabWindows/CVI includes a sample program called `ddedemo.prj` that uses DDE to send data to Microsoft Excel. The example program is located in the `samples\dde` directory. The following discussion outlines the process required to open an Excel worksheet file, send data over DDE, and set up a DDE link with one of the cells in the worksheet from a LabWindows/CVI program.

Start Excel and load the worksheet file called `lwcvi.xls`. The sample program performs the following operations:

1. Connects to the Microsoft Excel worksheet as a client.

   `ConnectToDDEServer`, with `excel` as the server name and `lwcvi.xls` as the topic name, establishes a connection with the worksheet. The client callback function `ClientCallback` identifies the function that processes the DDE transactions this particular conversation generates.

2. Establishes a DDE warm link with a particular cell in the Excel worksheet.

   `SetUpDDEWarmLink`, with the cell address `R5C2` as the item name, establishes a DDE link with the cell in the worksheet. Thereafter, whenever the value of cell `B5`—row 5, column 2—changes, Excel sends information to LabWindows/CVI by invoking the `clientCallback` function.

3. Sends data to the Excel worksheet from LabWindows/CVI.

   The program formats the data as a string and sends it to Excel using `ClientDDEWrite` with the Excel cell region `R1C2:R50C2` as the item name and a character array that contains 50 elements as the buffer pointer.

4. The callback function responds to DDE transactions from the Excel worksheet.

   The callback function automatically returns the following information:

   handle—Conversation that triggered the callback. One callback function can process multiple DDE conversations.

   item name—Cell(s) involved.

   topic name—Excel system or Excel file.

   transaction type—Either `DDE_DATAREADY` or `DDE_DISCONNECT`.

data format—`CF_TEXT` in this case.

data size—Number of bytes in the data.

data pointer—Pointer to the data.

callback data—User defined; `NULL` in this case.

When the callback function receives the `DDE_DATAREADY` transaction, it updates a numeric display by passing the data pointer value to a numeric control on the `.uir` file. When the DDE event is `DDE_DISCONNECT`, `DisconnectFromDDEServer` ends the DDE conversation.

Multithreading under Windows 95/NT

Although it is safe to use DDE Library functions in a multithreaded executable, you must observe a few restrictions. The following restrictions stem from limitations in the Windows implementation of DDE:

- After you call `RegisterDDEServer` to register your program as a server, you must make all subsequent function calls that apply to the server in the same thread in which you called `RegisterDDEServer`.
- After you call `ConnectToDDEServer` to create a client connection, you must make all subsequent functions calls that apply to the connection in the same thread in which you called `ConnectToDDEServer`.

DDE Library Function Reference

This section describes each function in the LabWindows/CVI DDE Library in alphabetical order.

AdviseDDEDataReady

```
int nbytes = AdviseDDEDataReady (unsigned int conversationHandle,
                                 char itemName[], unsigned int dataFormat,
                                 void *dataPointer, unsigned int dataSize,
                                 unsigned int timeout);
```

Purpose

This server function writes data to a DDE client application. Call `AdviseDDEDataReady` in your server program only when the value of a data item changes and a warm or hot link has been established for the data item.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
itemName	string	Uniquely identifies the output item; for example, an Excel cell name.
dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
dataPointer	void pointer	Pointer to buffer that holds data.
dataSize	unsigned integer	Number of bytes in data. Must be `0` if **dataPointer** is `NULL`. Limited to 64 KB under Windows 3.1 and Windows 95.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes written to the client. `<0` = error; refer to Table 6-3 for error codes

Parameter Discussion

dataFormat must be a valid data format Windows recognizes. Windows supports the following valid data formats:

CF_TEXT	CF_PALETTE
CF_BITMAP	CF_PENDATA
CF_METAFILEPICT	CF_RIFF
CF_SYLK	CF_WAVE
CF_DIF	CF_OWNERDISPLAY
CF_TIFF	CF_DSPTEXT
CF_OEMTEXT	CF_DSPBITMAP
CF_DIB	CF_DSPMETAFILEPICT

Refer to Microsoft programmer's documentation for Windows for an in-depth discussion of DDE programming and the meaning of each data format type.

Using This Function

AdviseDDEDataReady allows your program, acting as a DDE server, to send data to a client that has a hot or warm link.

When a client sets up a hot or warm link, your server callback function receives a DDE_ADVISELOOP message for a particular data object that corresponds to **itemName**. When the hot or warm link is terminated, your server callback function receives a DDE_ADVISESTOP message for the data object.

During the period when the hot or warm link is in effect, your server program is responsible for notifying the client whenever the value of the data object changes. When the data object value changes, you can call AdviseDDEDataReady or BroadcastDDEDataReady.

AdviseDDEDataReady differs from BroadcastDDEDataReady in that you specify a particular conversation with a client. AdviseDDEDataReady sends the data only to the client you specify with **conversationHandle**, even if other clients have hot or warm links to the same item. AdviseDDEDataReady sends the data without invoking your server callback function. However, if other clients have warm links to the same item, the function notifies them all that new data is available. If the clients request the new data, the DDE_REQUESTDATA message invokes your server callback function. If you do not want to send the data to those other clients, you must write your server callback function so that it does not call ServerDDEWrite in this case.

If you pass NULL (0) as **dataPointer** and 0 as **dataSize**, AdviseDDEDataReady sends no data to the client you specify with **conversationHandle**. Instead, the function notifies all clients with warm links to the item. If the clients request the new data, the DDE_REQUESTDATA message invokes your server callback function, and you can use ServerDDEWrite to send the data in response.

If successful, `AdviseDDEDataReady` returns the number of bytes sent. Otherwise, `AdviseDDEDataReady` returns a negative error code. Refer to Table 6-3 at the end of this chapter for error codes.

Note ***Your program should not call*** `AdviseDDEDataReady` ***in a tight loop because the iterations compete with user interface events for the CPU time. You should use*** `AdviseDDEDataReady` ***sparingly, and only when the value of the hot- or warm-linked data object changes. In cases when the server returns large data objects, your program should call*** `AdviseDDEDataReady` ***only when the user interface is not busy.***

See Also

`RegisterDDEServer`, `SetUpDDEHotLink`, `SetUpDDEWarmLink`, `BroadcastDDEDataReady`

BroadcastDDEDataReady

```
int nbytes = BroadcastDDEDataReady (char serverName[], char topicName[],
                                    char itemName[], unsigned int dataFormat,
                                    void *dataPointer, unsigned int dataSize)
```

Purpose

This server function sends data to all clients that have set up hot or warm links on **topicName** and **itemName**.

Parameters

Input

Name	Type	Description
serverName	string	Identifies the server from which to send the data.
topicName	string	Identifies the topic with which the data is associated.
itemName	string	Identifies the item with which the data is associated.
dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
dataPointer	void pointer	Pointer to buffer that holds data.
dataSize	unsigned integer	Number of bytes in data. Limited to 64 KB on Windows 3.1 and Windows 95.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes written to the client. `<0` = error; refer to Table 6-3 for error codes

Parameter Discussion

serverName, **topicName**, and **itemName** must be strings of length from 1 to 255. You can use them without regard to case.

Using This Function

`BroadcastDDEDataReady` allows your program, acting as a DDE server, to send data to all clients that have hot or warm links on the topic and item you specify.

When a client sets up a hot or warm link, your server callback function receives a `DDE_ADVISELOOP` message for a particular data object that corresponds to **itemName**. When the hot or warm link is terminated, your server callback function receives a `DDE_ADVISESTOP` message for the data object.

During the period when the hot or warm link is in effect, your server program is responsible for notifying the client whenever the value of the data object changes. When the data object value changes, your server program should call `BroadcastDDEDataReady` or `AdviseDDEDataReady`.

`BroadcastDDEDataReady` differs from `AdviseDDEDataReady` in that it is not restricted to a particular client. `BroadcastDDEDataReady` sends the data automatically to all clients with hot links to the item. `BroadcastDDEDataReady` notifies all clients with warm links to the item. For each warm-linked client that requests the data, the `DDE_REQUESTDATA` message invokes your server callback function. You must call `ServerDDEWrite` in the callback to send the data.

When successful, `BroadcastDDEDataReady` returns the number of bytes sent. Otherwise, `BroadcastDDEDataReady` returns a negative error code. Refer to Table 6-3 at the end of this chapter for error codes.

☞ **Note** ***Your program should not call*** `BroadcastDDEDataReady` ***in a tight loop because the iterations compete with user interface events for the CPU time. You should use*** `BroadcastDDEDataReady` ***sparingly and only when the value of the hot- or warm-linked data object changes. In cases when the server returns large data objects, your program should call*** `BroadcastDDEDataReady` ***only when the user interface is not busy.***

See Also

`RegisterDDEServer`, `SetUpDDEHotLink`, `SetUpDDEWarmLink`, `AdviseDDEDataReady`

ClientDDEExecute

```
int status = ClientDDEExecute (unsigned int conversationHandle,
                               char commandString[], unsigned int timeout);
```

Purpose

This client function sends a command for a DDE server application to execute.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
commandString	string	Command for the server application to execute.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
status	integer	Refer to Table 6-3 for error codes.

Parameter Discussion

commandString represents a valid command sequence for the server application to execute. Refer to the command function reference manual for the application to which you are connecting for more information on the commands supported.

See Also

`ConnectToDDEServer`, `ClientDDERead`, `ClientDDEWrite`

ClientDDERead

```
int nbytes = ClientDDERead (unsigned int conversationHandle,
                            char itemName[], unsigned int dataFormat,
                            void *dataBuffer, unsigned int dataSize,
                            unsigned int timeout);
```

Purpose

This client function reads data from a DDE server application.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
itemName	string	Uniquely identifies the output item; for example, an Excel cell name.
dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
dataSize	unsigned integer	Number of bytes to read. Limited to 64 KB under Windows 3.1 and Windows 95.
timeout	unsigned integer	Timeout in milliseconds.

Output

Name	Type	Description
dataBuffer	void pointer	Buffer in which to receive data.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes read from the server. `<0` = error; refer to Table 6-3 for error codes

Parameter Discussion

dataFormat must be a valid data format Windows recognizes. Windows supports the following data types:

CF_TEXT	CF_PALETTE
CF_BITMAP	CF_PENDATA
CF_METAFILEPICT	CF_RIFF
CF_SYLK	CF_WAVE
CF_DIF	CF_OWNERDISPLAY
CF_TIFF	CF_DSPTEXT
CF_OEMTEXT	CF_DSPBITMAP
CF_DIB	CF_DSPMETAFILEPICT

Refer to Microsoft programmer's documentation for Windows for an in-depth discussion of DDE programming and the meaning of each data format type.

status returns a positive number that represents the number of bytes that were successfully read. A negative number is an error code.

See Also

ConnectToDDEServer, ClientDDEWrite

ClientDDEWrite

```
int nbytes = ClientDDEWrite (unsigned int conversationHandle,
                             char itemName[], unsigned int dataFormat,
                             void *dataPointer, unsigned int dataSize,
                             unsigned int timeout);
```

Purpose

This client function writes data to a DDE server application.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
itemName	string	Uniquely identifies the output item; for example, an Excel cell name.
dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
dataPointer	void pointer	Buffer that holds data.
dataSize	unsigned integer	Number of bytes to write. Limited to 64 KB under Windows 3.1 and Windows 95.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes written to the server. `<0` = error; refer to Table 6-3 for error codes

Parameter Discussion

dataFormat must be a valid data format Windows recognizes. Windows supports the following valid data formats:

CF_TEXT	CF_PALETTE
CF_BITMAP	CF_PENDATA
CF_METAFILEPICT	CF_RIFF
CF_SYLK	CF_WAVE
CF_DIF	CF_OWNERDISPLAY
CF_TIFF	CF_DSPTEXT
CF_OEMTEXT	CF_DSPBITMAP
CF_DIB	CF_DSPMETAFILEPICT

Refer to Microsoft programmer's documentation for Windows for an in-depth discussion of DDE programming and the meaning of each data format type.

status returns a positive number that represents the number of bytes that were successfully read. A negative number is an error code.

See Also

ConnectToDDEServer, ClientDDERead

ConnectToDDEServer

```
int status = ConnectToDDEServer (unsigned int *conversationHandle,
                                 char serverName[], char topicName[],
                                 ddeFuncPtr clientCallbackFunction,
                                 void *callbackData);
```

Purpose

Establishes a connection, or conversation, between your program and a named server on a topic name you specify.

Parameters

Input

Name	Type	Description
serverName	string	Name of the server application.
topicName	string	Specifies the type of conversation with the server.
clientCallbackFunction	DDE function pointer	Pointer to the callback function that processes messages for the client.
callbackData	void pointer	User-defined data.

Output

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.

Return Value

Name	Type	Description
status	integer	Refer to Table 6-3 for error codes.

Parameter Discussion

conversationHandle returns an integer value that uniquely represents a conversation between a server and a client.

serverName and **topicName** must be strings of length from 1 to 255. You can use them without regard to case.

Each server application defines its own set of valid topic names. Refer to the command function reference manual for the server application. A client and a server can have multiple connections as long as they are under different topic names.

clientCallbackFunction is the name of a callback function to which the DDE Library sends messages from the server.

The callback function must be of the following form:

```
int (*ddeFuncPtr) (int handle, char *topicName, char *itemName,
                   int xType, int dataFmt, int dataSize,
                   void *dataPtr, void *callbackData);
```

xType specifies the type of message the server sends.

clientCallbackFunction can receive only two transaction types: `DDE_DISCONNECT` and `DDE_DATAREADY`.

`DDE_DISCONNECT`—Received when a server requests the termination of a connection or when Windows terminates the connection because of an internal error.

`DDE_DATAREADY`—Received when you have already set up a hot or warm link by calling `SetUpDDEHotLink` or `SetUpDDEWarmLink` and the server notifies you that new data is available. If the server program uses the LabWindows/CVI DDE Library, it notifies you by calling `AdviseDDEDataReady` or `BroadcastDDEDataReady`. The callback receives the data in **dataPtr**. **topicName**, **itemName**, **dataFmt**, **dataSize**, and **dataPtr** contain significant data. The server to which you are connecting assigns its own meaning to **topicName**. **itemName** can specify an object to which the data refers. For example, in Excel, the item name specifies a cell. **dataFmt** is one of the Windows-defined data types, for example, `CF_TEXT`. **dataSize** specifies the number of bytes in the data **dataPtr** points to.

Note ***The* dataSize *value is the value LabWindows/CVI receives from Windows. This value can be larger than the actual number of bytes the client writes.***

Note ***Return TRUE from the callback function if it can process the message successfully. Otherwise, return FALSE. The callback function should be short and return as soon as possible.***

callbackData is a 4-byte value the DDE Library passes to the callback function each time the DDE Library invokes the callback for the same client.

It is your responsibility to define the meaning of the callback data. For example, you can use the callback data as a pointer to a data object that you need to access in the callback function. In this way, you do not need to declare the data object as a global variable.

If you do not want to use the callback data, you can pass zero.

Note ***In the case of*** `DDE_DISCONNECT`***, the value of*** **callbackData** ***is undefined.***

See Also

`DisconnectFromDDEServer`, `RegisterDDEServer`

DisconnectFromDDEServer

```
int status = DisconnectFromDDEServer (unsigned int conversationHandle);
```

Purpose

Disconnects your client program from a server application.

Parameter

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.

Return Value

Name	Type	Description
status	integer	Refer to Table 6-3 for error codes.

Note `DisconnectFromDDEServer` ***ends only the client-server conversation that*** `conversationHandle` ***identifies. Multiple, concurrent conversations can exist between a client and a server.***

See Also

`ConnectToDDEServer`, `RegisterDDEServer`

GetDDEErrorString

```
char *message = GetDDEErrorString (int errorNum)
```

Purpose

Converts the error number that a DDE Library function returns into a meaningful error message.

Parameter

Input

Name	Type	Description
errorNum	integer	Error code a DDE function returns.

Return Value

Name	Type	Description
message	string	Explanation of error.

RegisterDDEServer

```
int status = RegisterDDEServer (char serverName[],
                                ddeFuncPtr serverCallbackFunction,
                                void *callbackData);
```

Purpose

Registers your program as a valid DDE server, allowing other Windows applications to connect to it for interprocess communication.

Parameters

Input

Name	Type	Description
serverName	string	Name of the server.
serverCallbackFunction	DDE function pointer	Pointer to the callback function that processes messages for the server.
callbackData	void pointer	Pointer to the user data.

Return Value

Name	Type	Description
status	integer	Refer to Table 6-3 for error codes.

Parameter Discussion

serverName must be a string of length from 1 to 255. You can use it without regard to case.

serverCallbackFunction is the name of a callback function that the DDE Library calls to process client requests.

The callback function must be of the following form:

```
int (*ddeFuncPtr) (int handle, char *topicName, char *itemName,
                   int xType, int dataFmt, int dataSize,
                   void *dataPtr, void *callbackData);
```

xType specifies the type of request the client sent. **serverCallbackFunction** can receive the following transaction types:

`DDE_CONNECT`—Received when a client requests a connection. **topicName** specifies the connection topic. The server defines the set of valid topic names and uses them in different ways. For example, Excel uses the topic name to specify the file the client asks to operate on.

A client can have multiple connections to the same server as long as a different topic name exists for each connection.

`DDE_DISCONNECT`—Received when a client requests the termination of a connection or when Windows terminates the connection because of an internal error.

`DDE_DATAREADY`—Received when the client sends data through DDE to the server. **topicName**, **itemName**, **dataFmt**, **dataSize**, and **dataPtr** contain significant data. **itemName** can specify an object to which the data refers. For example, in Excel, the item name specifies a cell. **dataFmt** is one of the Windows-defined data types, for example, `CF_TEXT`. **dataSize** specifies the number of bytes in the data **dataPtr** points to.

☞ **Note** ***The* dataSize *value is the value LabWindows/CVI receives from Windows. This value can be larger than the actual number of bytes the client writes.***

`DDE_REQUEST`—Received when the client requests that you send data to it through DDE. **itemName** can specify an object to which the data refers. For example, in Excel, the item name specifies a cell. **dataFmt** is one of the Windows-defined data types, for example, `CF_TEXT`.

`DDE_ADVISELOOP`—Received when the client requests a hot or warm link, or advisory loop, on a specific item. When a hot or warm link is in effect, the server is supposed to notify the client whenever the item you specify changes value. The server notifies the client of the change in value by calling `AdviseDDEDataReady` or `BroadcastDDEDataReady`. **itemName** and **dataFmt** contain significant values. **itemName** can specify an object to which the data item refers. For example, in Excel, the item name specifies a cell. **dataFmt** is one of the Windows-defined data types, for example, `CF_TEXT`.

`DDE_ADVISESTOP`—Received when the client requests the termination of an advisory loop. **itemName** contains the same value that the client used to set up the advisory loop.

`DDE_EXECUTE`—Received when the client requests the execution of a command. **itemName** contains the command string. The server defines the set of valid command strings. For example, Excel uses `"[Save()]"` to save a file.

Using This Function

`RegisterDDEServer` registers your program as a DDE server with the name you specify. Clients that attempt to connect to your program must use the specified name. Thereafter, `RegisterDDEServer` routes all client requests through the **serverCallbackFunction** you specify.

You can register your program as a DDE server multiple times as long as you specify different server names.

Note ***Return TRUE from the callback function if the request is successful. Otherwise, return FALSE. The callback function should be short and should return as soon as possible.***

callbackData is a 4-byte value the DDE Library passes to the callback function each time the DDE Library invokes the callback for the same server.

It is your responsibility to define the meaning of the callback data. The following examples show you how you can use the callback data:

- You can register your program as a DDE server multiple times under different names. For instance, you can use the same callback function for all the server instances by using the callback data to differentiate between them.
- You can use the callback data as a pointer to a data object that you need to access in the callback function. In this way, you do not need to declare the data object as a global variable.

If you do not want to use the callback data, you can pass zero.

Note ***In the case of*** `DDE_DISCONNECT`***, the value of*** **callbackData** ***is undefined.***

See Also

`ConnectToDDEServer`, `UnregisterDDEServer`

ServerDDEWrite

```
int nbytes = ServerDDEWrite (unsigned int conversationHandle,
                             char itemName[], unsigned int dataFormat,
                             void *dataPointer, unsigned int dataSize,
                             unsigned int timeout);
```

Purpose

Writes data to a DDE client application when the client requests data.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
itemName	string	Uniquely identifies the output item; for example, an Excel cell name.
dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
dataPointer	void pointer	Buffer that holds data.
dataSize	unsigned integer	Number of bytes to write. Limited to 64 KB under Windows 3.1 and Windows 95.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes written to the server. `<0` = error; refer to Table 6-3 for error codes

Parameter Discussion

dataFormat must be a valid data format Windows recognizes. Windows supports the following valid data formats:

CF_TEXT	CF_PALETTE
CF_BITMAP	CF_PENDATA
CF_METAFILEPICT	CF_RIFF
CF_SYLK	CF_WAVE
CF_DIF	CF_OWNERDISPLAY
CF_TIFF	CF_DSPTEXT
CF_OEMTEXT	CF_DSPBITMAP
CF_DIB	CF_DSPMETAFILEPICT

Refer to Microsoft programmer's documentation for Windows for an in-depth discussion of DDE programming and the meaning of each data format type.

Using This Function

ServerDDEWrite allows your program, acting as a DDE server, to send data to a client. You should call this function only when your **serverCallbackFunction** receives a DDE_REQUESTDATA transaction.

If you call the function at any other time, ServerDDEWrite stores the data until the client requests it. If you call the function multiple times on the same conversation before the client requests the data, ServerDDEWrite appends each new data set to the buffer that contains the stored data.

If the client has a hot or warm link and you need to send data *other than* in response to a DDE_REQUESTDATA transaction, use AdviseDDEDataReady or BroadcastDDEDataReady.

If successful, ServerDDEWrite returns the number of bytes written. Otherwise, ServerDDEWrite returns a negative error code.

See Also

RegisterDDEServer, AdviseDDEDataReady

SetUpDDEHotLink

```
int status = SetUpDDEHotLink (unsigned int conversationHandle,
                              char itemName[], unsigned int dataFormat,
                              unsigned int timeout);
```

Purpose

This client function sets up a hot link, or advisory loop, between the client and the server. `SetUpDDEHotLink` returns zero for success and a negative error code for failure.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
itemName	string	Uniquely identifies the output item; for example, an Excel cell name.
dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
status	integer	Refer to Table 6-3 for error codes.

Parameter Discussion

itemName identifies the information in the server application to which the DDE link applies. For example, the item name can represent an Excel range of cells by using the range description `R1C1:R10C10`.

Note ***To the client, LabWindows/CVI does not distinguish between a hot link and a warm link. For both types of links, the DDE Library calls the client callback function with a transaction type of*** `DDE_DATAREADY` ***when the data item changes at the server site. The new data is available in the*** **dataPtr** ***parameter of the callback function. LabWindows/CVI has two different functions for setting up a warm link or hot link in case some applications accept only one or the other kind of link.***

See Also

`RegisterDDEServer`, `SetUpDDEWarmLink`

SetUpDDEWarmLink

```
int status = SetUpDDEWarmLink (unsigned int conversationHandle,
                               char itemName[], unsigned int dataFormat,
                               unsigned int timeout);
```

Purpose

This client function sets up a warm link, or advisory loop, between the client and the server. `SetUpDDEWarmLink` returns zero for success and a negative error code for failure.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
itemName	string	Uniquely identifies the output item; for example, an Excel cell name.
dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
status	integer	Refer to Table 6-3 for error codes.

Parameter Discussion

itemName identifies the information in the server application to which the DDE link applies. For example, the item name can represent an Excel range of cells by using the range description `R1C1:R10C10`.

Note ***To the client, LabWindows/CVI does not distinguish between a hot link and a warm link. For both types of links, the DDE Library calls the client callback function with a transaction type of*** `DDE_DATAREADY` ***when the data item changes at the server site. The new data is available in the*** **dataPtr** ***parameter of the callback function. LabWindows/CVI has two different functions for setting up a warm link or hot link in case some applications accept only one or the other kind of link.***

See Also

`RegisterDDEServer`, `SetUpDDEHotLink`

TerminateDDELink

```
int status = TerminateDDELink (unsigned int conversationHandle,
                               char itemName[], unsigned int dataFormat,
                               unsigned int timeout);
```

Purpose

Lets your program, acting as a DDE client, terminate an advisory loop you previously established with the server through `SetUpDDEWarmLink` or `SetUpDDEHotLink`.

`TerminateDDELink` returns zero for success or a negative error code for failure.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
itemName	string	Uniquely identifies the output item; for example, an Excel cell name.
dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
status	integer	Refer to Table 6-3 for error codes.

UnregisterDDEServer

```
int status = UnregisterDDEServer (char serverName[]);
```

Purpose

Unregisters your application program as a DDE server.

Parameter

Input

Name	Type	Description
serverName	string	Name of the server.

Return Value

Name	Type	Description
status	integer	Refer to Table 6-3 for error codes.

See Also

RegisterDDEServer

Error Conditions

If an error condition occurs during a call to any of the functions in the LabWindows/CVI DDE Library, the status return value contains the error code. This code is a nonzero value that specifies the type of error that occurred. Error codes are negative numbers. Table 6-3 lists the currently defined error codes and the associated meanings.

Table 6-3. DDE Library Error Codes

Code	Error Message
0	kDDE_NoError
-1	-kDDE_UnableToRegisterService
-2	-kDDE_ExistingServer
-3	-kDDE_FailedToConnect
-4	-kDDE_ServerNotRegistered
-5	-kDDE_TooManyConversations
-9	-kDDE_InvalidParameter
-10	-kDDE_OutOfMemory
-12	-kDDE_NoConnectionEstablished
-13	-kDDE_NotThreadOfServer
-14	-kDDE_NotThreadOfClient
-16	-kDDE_AdvAckTimeOut
-17	-kDDE_Busy
-18	-kDDE_DataAckTimeOut
-19	-kDDE_DllNotInitialized
-20	-kDDE_DllUsage
-21	-kDDE_ExecAckTimeOut
-22	-kDDE_DataMismatch
-23	-kDDE_LowMemory
-24	-kDDE_MemoryError
-25	-kDDE_NotProcessed
-26	-kDDE_NoConvEstablished

Table 6-3. DDE Library Error Codes (Continued)

Code	Error Message
-27	-kDDE_PokeAckTimeOut
-28	-kDDE_PostMsgFailed
-30	-kDDE_ServerDied
-31	-kDDE_SysError
-32	-kDDE_UnadvAckTimeOut
-33	-kDDE_UnfoundQueueId

Note ***Error codes*** `-16` ***to*** `-33` ***correspond to native Windows DDE error codes that start from*** `0x4000`**.**

7

TCP Library

This chapter describes the functions in the LabWindows/CVI TCP (Transmission Control Protocol) Library. The *TCP Library Function Overview* section contains general information about the TCP Library functions and panels. The *TCP Library Function Reference* section contains an alphabetical list of function descriptions.

To use this library under Windows, you must have a version of `winsock.dll`. The DLL comes with the program that drives the network card.

TCP Library Function Overview

This section contains general information about the TCP Library functions and network communication using TCP. TCP Library functions provide a platform-independent interface to the reliable, connection-oriented, byte-stream, network communication protocol.

TCP Library Function Panels

The TCP Library function panels are grouped in the tree structure in Table 7-1 according to the types of operations they perform.

The first- and second-level headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels. Each TCP Library function panel generates one TCP Library function call.

Table 7-1. Functions in the TCP Library Function Tree

Class/Panel Name	Function Name
Server Functions	
Register TCP Server	`RegisterTCPServer`
Server TCP Read	`ServerTCPRead`
Server TCP Write	`ServerTCPWrite`
Unregister TCP Server	`UnregisterTCPServer`
Disconnect TCP Client	`DisconnectTCPClient`
Client Functions	
Connect to TCP Server	`ConnectToTCPServer`
Client TCP Read	`ClientTCPRead`
Client TCP Write	`ClientTCPWrite`
Disconnect from TCP Server	`DisconnectFromTCPServer`

Table 7-1. Functions in the TCP Library Function Tree (Continued)

Class/Panel Name	Function Name
Support Functions	
Get Host IP Address	`GetTCPHostAddr`
Get Host Machine Name	`GetTCPHostName`
Get Peer IP Address	`GetTCPPeerAddr`
Get Peer Machine Name	`GetTCPPeerName`
Get System Socket Handle	`GetHostTCPSocketHandle`
Set Disconnect Mode	`SetTCPDisconnectMode`
Get Error String	`GetTCPErrorString`
Get System Error String	`GetTCPSystemErrorString`

The online help with each panel contains specific information about operating each function panel.

TCP Clients and Servers

Network communication using the TCP Library involves a client and a server in each connection. A TCP server can send and receive information to and from a client application through a network. A TCP client can send and request data to and from a server application. Once registered, a server waits for clients to request connection to it. A client, however, can request connection only to a pre-existing server.

With the LabWindows/CVI TCP Library, you can write programs to act as a TCP client or server. Under Windows, you cannot run both a server and a client on the same computer. The procedure for writing a program using TCP is similar to the procedure you follow for using DDE. Refer to the sample program discussion in Chapter 6, *DDE Library*. Two additional sample programs, `tcpserv.prj` and `tcpclnt.prj`, provide some guidelines on structuring your TCP programs as a server or a client. These programs are provided as templates only and require modification for operation on your computer.

To connect to a TCP server from a LabWindows/CVI program, you must have some information about the application to which you would like to connect. All TCP server applications must run on a specified host, which has a known host name, for example, `aaa.bbb.ccc`, or a known IP address, for example, `123.456.78.90`, associated with it. In addition, each server has a unique port number on the host computer. These two pieces of information identify different servers on the same computer or on different computers. Before any client program can connect to a server, it must know the host name and server port number.

If you want your program to act as a TCP server, you must call `RegisterTCPServer` in your program. `RegisterTCPServer` establishes your program as the server associated with a port number on the local host. Client applications can connect to your program by using the port number associated with the server and either the host name or the IP address of the computer

on which the server application is currently running. The TCP Library calls your server callback function whenever the conversation partner requests communication.

TCP Callback Function

Callback functions provide the mechanism for receiving notification of connection, connection termination, and data availability. Similar to the method in which a callback function responds to user interface events from your User Interface Library object files, a TCP callback function responds to incoming TCP messages and information.

A callback function can respond to three types of TCP messages: `TCP_CONNECT`, `TCP_DISCONNECT`, and `TCP_DATAREADY`.

If your program acts as a TCP server, client applications can trigger your TCP callback function in a number of ways. Whenever a client application attempts to connect to your server program or requests information from your program, the TCP Library invokes your callback function to process the request.

The parameter prototype for the TCP callback function in LabWindows/CVI is defined as follows:

```
int CallbackFunction (int handle, int xType, int errCode,
                      void *callbackData);
```

where **handle** represents the conversation handle

xType represents the transaction type; refer to Table 7-2 in this section

errCode for `TCP_DISCONNECT` is negative if the connection terminates because of an error

callbackData is a user-defined data value

Table 7-2 lists the TCP transaction types, **xType**, that can trigger a callback function.

Table 7-2. TCP Transaction Types (xType)

xType	Server	Client	When?
`TCP_CONNECT`	Y	N	When a new client requests connection.
`TCP_DISCONNECT`	Y	Y	When conversation partner quits.
`TCP_DATAREADY`	Y	Y	When conversation partner sends data.

Refer to `RegisterTCPServer` and `ConnectToTCPServer` for more information about the TCP callback function.

Multithreading under Windows 95/NT

It is safe to use TCP Library in a multithreaded executable. No restrictions exist.

TCP Library Function Reference

This section describes each function in the LabWindows/CVI TCP Library in alphabetical order.

ClientTCPRead

```
int nbytes = ClientTCPRead (unsigned int conversationHandle,
                            void *dataBuffer, unsigned int dataSize,
                            unsigned int timeout);
```

Purpose

Reads data from a TCP server application.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
dataBuffer	void pointer	Buffer in which to receive data.
dataSize	unsigned integer	Maximum number of bytes to read.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes read from the server. `<0` = error; refer to Table 7-3 for error codes

See Also

`ConnectToTCPServer`, `ClientTCPWrite`

ClientTCPWrite

```
int nbytes = ClientTCPWrite (unsigned int conversationHandle,
                             void *dataPointer, int dataSize,
                             unsigned int timeout);
```

Purpose

Writes data to a TCP server application.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
dataPointer	void pointer	Buffer that holds data.
dataSize	unsigned integer	Number of bytes to write.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes written to the server. `<0` = error; refer to Table 7-3 for error codes

See Also

`ConnectToTCPServer`, `ClientTCPRead`

ConnectToTCPServer

```
int status = ConnectToTCPServer (unsigned int *conversationHandle,
                                 unsigned int portNumber, char *serverHostName,
                                 tcpFuncPtr clientCallbackFunction,
                                 void *callbackData, unsigned int timeout);
```

Purpose

Establishes a conversation between your program and a pre-existing server. Your program becomes a client.

Parameters

Input

Name	Type	Description
portNumber	unsigned integer	Uniquely identifies a server on a single computer.
serverHostName	string	Host name or IP address of the server computer; for example, `aaa.bbb.ccc` or `123.456.78.90`.
clientCallbackFunction	TCP function pointer	Pointer to the callback function that processes messages for the client.
callbackData	void pointer	User-defined data.
timeout	unsigned integer	Timeout in milliseconds.

Output

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

Parameter Discussion

clientCallbackFunction is the name of the function the TCP Library calls to process messages your program receives as a TCP client.

The callback function must be of the following form:

```
int (*tcpFuncPtr) (int handle, int xType, int errCode,
                   void *callbackData);
```

xType specifies the type of message the server sends. The client callback function can receive the following transaction types:

`TCP_DISCONNECT`

`TCP_DATAREADY`

Use **errCode** only when the transaction type is `TCP_DISCONNECT`.

`TCP_DISCONNECT`—Received when a server requests the termination of a connection or when a connection terminates because of an error. If the connection terminates because of an error, the **errCode** parameter contains a negative error code. Refer to Table 7-3 at the end of this chapter for the list of error codes.

`TCP_DATATREADY`—Received when the server sends data through TCP to the client. Your program, acting as the client, calls `ClientTCPRead` to obtain the data.

☞ **Note** ***Return TRUE for the client callback function if it can process the message successfully. Otherwise, return FALSE. The callback function should be short and should return as soon as possible.***

callbackData is a 4-byte value the TCP Library passes to the callback function each time the TCP Library invokes the callback for the same client.

It is your responsibility to define the meaning of the callback data. One way to use **callbackData** is as a pointer to a data object that you need to access in the callback function. By doing this, you can avoid declaring the data object as a global variable.

If you do not want to use **callbackData**, you can pass zero.

See Also

`RegisterTCPServer`, `DisconnectFromTCPServer`

DisconnectFromTCPServer

```
int status = DisconnectFromTCPServer (unsigned int conversationHandle);
```

Purpose

Disconnects your client program from a server application.

Parameter

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

Note `DisconnectFromTCPServer` ***terminates only the connection*** **conversationHandle** ***identifies. There can be more than one conversation between a client and a server.***

See Also

`ConnectToTCPServer`, `RegisterTCPServer`, `SetTCPDisconnectMode`

DisconnectTCPClient

```
int status = DisconnectTCPClient (unsigned int conversationHandle);
```

Purpose

This TCP server function terminates a connection with a client.

Parameter

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the connection.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

Note `DisconnectTCPClient` ***terminates only the connection*** **conversationHandle** ***identifies. There can be more than one conversation between a client and a server.***

See Also

`RegisterTCPServer`, `SetTCPDisconnectMode`

GetHostTCPSocketHandle

```
int status = GetHostTCPSocketHandle (unsigned int connectionHandle,
                                     int systemSocket);
```

Purpose

Obtains the system socket handle that corresponds to a TCP Library connection.

Parameters

Input

Name	Type	Description
connectionHandle	unsigned integer	TCP Library conversation handle you obtain from `ConnectToTCPServer` or receive in a server callback as the handle parameter of a `TCP_DISCONNECT` message.

Output

Name	Type	Description
systemSocket	unsigned integer	System socket handle for the connection `connectionHandle` identifies.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

GetTCPErrorString

```
char *message = GetTCPErrorString (int errorNum);
```

Purpose

Converts the error number a TCP Library function returns into a meaningful error message.

Parameter

Input

Name	Type	Description
errorNum	integer	Status value a TCP function returns.

Return Value

Name	Type	Description
message	string	Explanation of error.

See Also

GetTCPSystemErrorString

GetTCPHostAddr

```
int status = GetTCPHostAddr (char buffer[], int bufferSize);
```

Purpose

Obtains the IP address of the computer on which your program is currently running.

Parameters

Input

Name	Type	Description
bufferSize	integer	Number of bytes in **buffer**, including space for the ASCII NUL byte.

Output

Name	Type	Description
buffer	character array	Buffer `GetTCPHostAddr` fills with the IP address of the computer your program is currently running on.

Parameter Discussion

`GetTCPHostAddr` fills in **buffer** with the IP address of your computer. The IP address is in the dot format, as in `"130.164.1.1"`.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

GetTCPHostName

```
int status = GetTCPHostName (char buffer[], int bufferSize);
```

Purpose

Obtains the name of the computer on which your program is currently running.

Parameters

Input

Name	Type	Description
bufferSize	integer	Number of bytes in **buffer**, including space for the ASCII NUL byte.

Output

Name	Type	Description
buffer	character array	Buffer `GetTCPHostName` fills with the name of the computer your program is currently running on.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

GetTCPPeerAddr

```
int status = GetTCPPeerAddr (unsigned int connectionHandle,
                             char buffer[], int bufferSize);
```

Purpose

Obtains the IP address of the computer on which a remote client or server is currently running.

Parameters

Input

Name	Type	Description
connectionHandle	unsigned integer	TCP Library conversation handle you obtain from `ConnectToTCPServer` or receive in a server callback as the handle parameter of a `TCP_CONNECT` message.
bufferSize	integer	Number of bytes in **buffer**, including space for the ASCII NUL byte.

Output

Name	Type	Description
buffer	character array	Buffer `GetTCPPeerAddr` fills with the IP address of the computer the remote client or server program is currently running on.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

Parameter Discussion

`GetTCPPeerAddr` fills in **buffer** with the IP address of the remote client or server computer. The IP address is in the dot format, as in `"130.164.1.1"`.

GetTCPPeerName

```
int status = GetTCPPeerName (unsigned int connectionHandle,
                             char buffer[], int bufferSize);
```

Purpose

Obtains the name of the computer on which a remote client or server is currently running.

Parameters

Input

Name	Type	Description
connectionHandle	unsigned integer	TCP Library conversation handle you obtain from `ConnectToTCPServer` or receive in a server callback as the handle parameter of a `TCP_CONNECT` message.
bufferSize	integer	Number of bytes in **buffer**, including space for the ASCII NUL byte.

Output

Name	Type	Description
buffer	character array	Buffer `GetTCPPeerName` fills with the name of the computer the remote client or server program is currently running on.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

GetTCPSystemErrorString

```
char *errorMsgString = GetTCPSystemErrorString (void);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `GetTCPSystemErrorString`**.**

Purpose

Obtains a system message that describes the error that caused a TCP Library function to fail. The messages `GetTCPSystemErrorString` returns are sometimes more descriptive than the error codes the TCP Library functions return.

To obtain the correct system error message, you must call `GetTCPSystemErrorString` immediately after you call the TCP Library function that failed.

Parameters

None.

Return Value

Name	Type	Description
errorMsgString	string	System error message string.

See Also

`GetTCPErrorString`

RegisterTCPServer

```
int status = RegisterTCPServer (unsigned int portNumber,
                                tcpFuncPtr serverCallbackFunction,
                                void *callbackData);
```

Purpose

Registers your program as a valid TCP server and allows other applications to connect to it for network communication.

Parameters

Input

Name	Type	Description
portNumber	unsigned integer	Uniquely identifies a server on a single computer.
serverCallbackFunction	TCP function pointer	Pointer to the callback function that processes messages for the server.
callbackData	void pointer	Pointer to the user data.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

Parameter Discussion

serverCallbackFunction is the name of the function the TCP Library calls to process client requests.

The callback function must be of the following form:

```
int (*tcpFuncPtr) (int handle, int xType, int errCode,
                   void *callbackData);
```

xType specifies the type of message the server sends. The server callback function can receive the following transaction types:

`TCP_CONNECT`—Received when a client requests a connection.

`TCP_DISCONNECT`—Received when a client requests the termination of a connection or when a connection terminates because of an error. If the connection terminates because of an error,

the **errCode** parameter contains a negative error code. Refer to Table 7-3 at the end of this chapter for error codes.

`TCP_DATATREADY`—Received when the client sends data through TCP to the server. Your program, acting as the server, calls `ServerTCPRead` to obtain the data.

Use **errCode** only when the transaction type is `TCP_DISCONNECT`.

Note ***Return TRUE for the server callback function if the request is successful. Otherwise, return FALSE. Server callbacks should be short and should return as soon as possible.***

callbackData is a 4-byte value the TCP Library passes to the callback function each time the TCP Library invokes the callback for the same server.

It is your responsibility to define the meaning of the callback data. The following examples show you how you can use the callback data:

- You can register your program as a TCP server multiple times under different port numbers. You can use the same callback function for all the server instances by using the callback data to differentiate between them.
- You can use the callback data as a pointer to a data object that you need to access in the callback function. By doing this, you can avoid declaring the data object as a global variable.

If you do not want to use the callback data, you can pass zero.

See Also

`ConnectToTCPServer`, `UnregisterTCPServer`

ServerTCPRead

```
int nbytes = ServerTCPRead (unsigned int conversationHandle,
                            void *dataBuffer, unsigned int dataSize,
                            unsigned int timeout);
```

Purpose

Reads data from a TCP client application.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
dataBuffer	void pointer	Buffer in which to receive data.
dataSize	unsigned integer	Number of bytes to read.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes read from the client. `<0` = error; refer to Table 7-3 for error codes

See Also

`RegisterTCPServer`, `ServerTCPWrite`

ServerTCPWrite

```
int nbytes = ServerTCPWrite (unsigned int conversationHandle,
                             void *dataPointer, unsigned int dataSize,
                             unsigned int timeout);
```

Purpose

Writes data to a TCP client application.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	Uniquely identifies the conversation.
dataPointer	void pointer	Buffer that holds data.
dataSize	unsigned integer	Number of bytes to write.
timeout	unsigned integer	Timeout in milliseconds.

Return Value

Name	Type	Description
nbytes	integer	Number of bytes written to the client. `<0` = error; refer to Table 7-3 for error codes

See Also

`RegisterTCPServer`, `ServerTCPRead`

SetTCPDisconnectMode

```
int status = SetTCPDisconnectMode (unsigned int conversationHandle,
                                   int disconnectMode);
```

Purpose

Tells the TCP Library whether to close the local conversation handle when a remote client or server terminates a connection. There are two modes: auto disconnect and manual disconnect.

In the auto disconnect mode (`TCP_DISCONNECT_AUTO`), the TCP Library closes the local conversation handle after it invokes your client or server callback with the `TCP_DISCONNECT` message. If, however, the library invokes the `TCP_DISCONNNECT` message while your program is already nested in a `ServerTCPRead` or `ClientTCPRead` call on the same conversation handle, the library does not close the conversation handle until the `ServerTCPRead` or `ClientTCPRead` function completes.

In the manual disconnect mode (`TCP_DISCONNECT_MANUAL`), the TCP Library never closes the local conversation handle automatically. You must call `DisconnectFromTCPServer` or `DisconnectTCPClient` to close the conversation handle. This method allows you to read data from the connection after you receive a `TCP_DISCONNECT` for the connection. You should close the handle to the connection after you have read all the data.

If you do not call `SetTCPDisconnectMode`, the TCP Library uses the auto disconnect method.

Parameters

Input

Name	Type	Description
conversationHandle	unsigned integer	TCP Library conversation handle you obtain from `ConnectToTCPServer` or receive in a server callback as the handle parameter of a `TCP_CONNECT` message.
disconnectMode	integer	Tells the TCP Library whether to close the conversation handle automatically. Use `TCP_DISCONNECT_AUTO` or `TCP_DISCONNECT_MANUAL`.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

See Also

`DisconnectFromTCPServer`, `DisconnectTCPClient`

UnregisterTCPServer

```
int status = UnregisterTCPServer (unsigned int portNumber);
```

Purpose

Unregisters your server application program as a TCP server.

Parameter

Input

Name	Type	Description
portNumber	unsigned integer	Uniquely identifies a server on a single computer.

Return Value

Name	Type	Description
status	integer	Refer to Table 7-3 for error codes.

See Also

RegisterTCPServer

Error Conditions

If an error condition occurs during a call to any of the functions in the LabWindows/CVI TCP Library, the status return value contains the error code. This code is a nonzero value that specifies the type of error that occurred. Error codes are negative numbers. Table 7-3 lists the currently defined error codes and the associated meanings.

Table 7-3. TCP Library Error Codes

Code	Error Message
0	kTCP_NoError
-1	-kTCP_UnableToRegisterService
-2	-kTCP_UnableToEstablishConnection
-3	-kTCP_ExistingServer
-4	-kTCP_FailedToConnect
-5	-kTCP_ServerNotRegistered
-6	-kTCP_TooManyConnections
-7	-kTCP_ReadFailed
-8	-kTCP_WriteFailed
-9	-kTCP_InvalidParameter
-10	-kTCP_OutOfMemory
-11	-kTCP_TimeOutErr
-12	-kTCP_NoConnectionEstablished
-13	-kTCP_GeneralIOErr
-14	-kTCP_ConnectionClosed
-15	-kTCP_UnableToLoadWinsockDLL
-16	-kTCP_IncorrectWinsockDLLVersion
-17	-kTCP_NetworkSubsystemNotReady
-18	-kTCP_ConnectionsStillOpen

8

Utility Library

This chapter describes the functions in the LabWindows/CVI Utility Library. The Utility Library contains functions that do not fit into any of the other LabWindows/CVI libraries. The *Utility Library Function Overview* section contains general information about the Utility Library functions and panels. The *Utility Library Function Reference* section contains an alphabetical list of function descriptions.

Utility Library Function Overview

The Utility Library function panels are grouped in the tree structure in Table 8-1 according to the type of operations they perform.

The first- and second-level headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels.

The following shows the structure of the ActiveX Automation Library function tree.

Timer/Wait
Date/Time
Keyboard
File Utilities
Directory Utilities
External Modules
Port I/O
Standard Input/Output Window
Run-Time Error Reporting
    Old-Style Functions
Interrupts
Physical Memory Access
Persistent Variable
Task Switching
Launching Executables
    Extended Functions
    Miscellaneous

Table 8-1. Functions in the Utility Library Function Tree

Class/Panel Name	Function Name
Timer/Wait	
Timer	`Timer`
Delay	`Delay`
Synchronized Wait	`SyncWait`
Date/Time	
Date in ASCII Format	`DateStr`
Time in ASCII Format	`TimeStr`
Get System Date	`GetSystemDate`
Set System Date	`SetSystemDate`
Get System Time	`GetSystemTime`
Set System Time	`SetSystemTime`
Keyboard	
Key Hit?	`KeyHit`
Get a Keystroke	`GetKey`
File Utilities	
Delete File	`DeleteFile`
Rename File	`RenameFile`
Copy File	`CopyFile`
Get File Size	`GetFileSize`
Get File Date	`GetFileDate`
Set File Date	`SetFileDate`
Get File Time	`GetFileTime`
Set File Time	`SetFileTime`
Get File Attributes	`GetFileAttrs`
Set File Attributes	`SetFileAttrs`
Get First File	`GetFirstFile`
Get Next File	`GetNextFile`
Make Pathname	`MakePathname`
Split Path	`SplitPath`
Directory Utilities	
Get Directory	`GetDir`
Get Project Directory	`GetProjectDir`
Get Module Directory	`GetModuleDir`
Get Full Path from Project	`GetFullPathFromProject`
Set Directory	`SetDir`
Make Directory	`MakeDir`
Delete Directory	`DeleteDir`
Get Drive	`GetDrive`
Set Drive	`SetDrive`
External Modules	
Load External Module	`LoadExternalModule`
Load External Module Ex	`LoadExternalModuleEx`
Run External Module	`RunExternalModule`
Get External Module Address	`GetExternalModuleAddr`

Table 8-1. Functions in the Utility Library Function Tree (Continued)

Class/Panel Name	Function Name
External Modules (continued)	
Unload External Module	`UnloadExternalModule`
Release External Module	`ReleaseExternalModule`
Port I/O	
Input Byte from Port	`inp`
Input Word from Port	`inpw`
Output Byte to Port	`outp`
Output Word to Port	`outpw`
Standard Input/Output Window	
Clear Screen	`Cls`
Get Stdio Window Options	`GetStdioWindowOptions`
Set Stdio Window Options	`SetStdioWindowOptions`
Get Stdio Window Position	`GetStdioWindowPosition`
Set Stdio Window Position	`SetStdioWindowPosition`
Get Stdio Window Size	`GetStdioWindowSize`
Set Stdio Window Size	`SetStdioWindowSize`
Get Stdio Window Visibility	`GetStdioWindowVisibility`
Set Stdio Window Visibility	`SetStdioWindowVisibility`
Get Stdio Port	`GetStdioPort`
Set Stdio Port	`SetStdioPort`
Run-Time Error Reporting	
Set Break on Library Errors	`SetBreakOnLibraryErrors`
Get Break on Library Errors	`GetBreakOnLibraryErrors`
Set Break on Protection Errors	`SetBreakOnProtectionErrors`
Get Break on Protection Errors	`GetBreakOnProtectionErrors`
Old-Style Functions	
Enable Break on Library Errors	`DisableBreakOnLibraryErrors`
Disable Break on Library Errors	`EnableBreakOnLibraryErrors`
Interrupts	
Disable Interrupts	`DisableInterrupts`
Enable Interrupts	`EnableInterrupts`
Get Interrupt State	`GetInterruptState`
Physical Memory Access	
Read from Physical Memory	`ReadFromPhysicalMemory`
Read from Physical Memory Ex	`ReadFromPhysicalMemoryEx`
Write to Physical Memory	`WriteToPhysicalMemory`
Write to Physical Memory Ex	`WriteToPhysicalMemoryEx`
Map Physical Memory	`MapPhysicalMemory`
Unmap Physical Memory	`UnMapPhysicalMemory`
Persistent Variable	
Set Persistent Variable	`SetPersistentVariable`
Get Persistent Variable	`GetPersistentVariable`
Task Switching	
Disable Task Switching	`DisableTaskSwitching`
Enable Task Switching	`EnableTaskSwitching`

Table 8-1. Functions in the Utility Library Function Tree (Continued)

Class/Panel Name	Function Name
Launching Executables	
Launch Executable	`LaunchExecutable`
Extended Functions	
Launch Executable Extended	`LaunchExecutableEx`
Has Executable Terminated?	`ExecutableHasTerminated`
Terminate Executable	`TerminateExecutable`
Retire Executable Handle	`RetireExecutableHandle`
Miscellaneous	
System Help	`SystemHelp`
Get CVI Version	`GetCVIVersion`
Get Current Platform	`GetCurrentPlatform`
In Standalone Executable?	`InStandaloneExecutable`
Run-Time Engine Detached?	`CVIRTEHasBeenDetached`
Initialize CVI Run-Time Engine	`InitCVIRTE`
Close CVI Run-Time Engine	`CloseCVIRTE`
Low-Level Support Driver Loaded	`CVILowLevelSupportDriverLoaded`
Beep	`Beep`
Breakpoint	`Breakpoint`
Round Real to Nearest Integer	`RoundRealToNearestInteger`
Truncate Real Number	`TruncateRealNumber`
Get Window Display Setting	`GetWindowDisplaySetting`
Check for Duplicate Instance	`CheckForDuplicateAppInstance`

Class Descriptions

- Timer/Wait functions use the system timer, including functions that wait on a timed basis.
- Date/Time functions return the date or time in ASCII or integer formats and set the date or time.
- Keyboard functions provide access to user keystrokes.
- File Utilities functions manipulate files.
- Directory Utilities functions manipulate directories and disk drives.
- External Modules functions load, execute, and unload files that contain compiled code.
- Port I/O functions read and write data from I/O ports; available only under Windows.
- Standard Input/Output Window functions control various attributes of the Standard Input/Output window.
- Run-Time Error Reporting functions enable and disable the features that break execution when certain types of errors occur.
- Interrupts functions disable and enable the occurrence of interrupts.
- Physical Memory Access functions read and write data from and to physical memory addresses; supported only under Windows.
- Persistent Variable functions store and retrieve an integer value across multiple builds and executions of a project in the LabWindows/CVI development environment.
- Task Switching functions control whether a user can switch to another task when running your program under Windows.
- Launching Executables functions start another executable, check whether it is still running, and terminate it.
- Miscellaneous functions perform a variety of operations that do not fit into any of the other function classes.

The online help with each panel contains specific information about operating each function panel.

Utility Library Function Reference

This section describes each of the functions in the LabWindows/CVI Utility Library in alphabetical order.

Beep

```
void Beep (void);
```

Purpose

Sounds the speaker.

Parameters

None.

Return Value

None.

Breakpoint

```
void Breakpoint (void);
```

Purpose

Suspends program operation. While the program is suspended, you can inspect or modify variables and use many other features of the LabWindows/CVI interactive program.

Calling `Breakpoint` with the debugging level set to None or from a compiled module has no effect.

Parameters

None.

Return Value

None.

CheckForDuplicateAppInstance

```
int status = CheckForDuplicateAppInstance (int activateOtherInstance,
                                           int *thereIsAnotherInstance);
```

Purpose

Determines if another copy of the same executable is running, but only if the other copy has already called this function. You can pass `ACTIVATE_OTHER_INSTANCE` in **activateOtherInstance** to bring the other copy to the front.

Use `CheckForDuplicateAppInstance` to prevent two instances of your application from running at the same time.

Note ***Only the Windows 95/NT versions of LabWindows/CVI support*** `CheckForDuplicateAppInstance`

For other platforms, `CheckForDuplicateAppInstance` always returns `-1`.

Parameters

Input

Name	Type	Description
activateOtherInstance	integer	Specifies whether to bring the other application instance to the front. Valid values are `ACTIVATE_OTHER_INSTANCE` or `DO_NOT_ACTIVATE_OTHER_INSTANCE`

Output

Name	Type	Description
thereIsAnotherInstance	integer	`1` if another instance of this executable exists; otherwise, `0`. If `CheckForDuplicateAppInstance` returns an error code, this variable is always set to zero.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
0	Success.
-1	You are running on a platform other than Windows 95/NT.
-2	Could not allocate system resources needed to check for a duplicate application instance.

Example

```
#include <cvirte.h>
int main (int argc, char *argv[])
{
   int thereIsAnother;
   if (InitCVIRTE (0, argv, 0) == 0)
      return -1;    /* out of memory */
   if (CheckForDuplicateAppInstance (ACTIVATE_OTHER_INSTANCE,
                                     &thereIsAnother) < 0)
      return -1;  /* out of memory, or not Win 95/NT */
   if (thereIsAnother)
      return 0;     /* prevent duplicate instance */
   return 0;
}
```

CloseCVIRTE

```
void CloseCVIRTE (void);
```

Purpose

Releases memory that `InitCVIRTE` allocated in the LabWindows/CVI Run-time Engine for a particular DLL.

If you call `InitCVIRTE` from `DllMain`, you also should call `CloseCVIRTE` from `DllMain`. You should call it in response to the `DLL_PROCESS_DETACH` message just before you return from `DllMain`.

Parameters

None.

Return Value

None.

Cls

```
void Cls (void);
```

Purpose

Clears the Standard I/O window.

Parameters

None.

Return Value

None.

CopyFile

```
int result = CopyFile (char sourceFileName[], char targetFileName[]);
```

Purpose

Copies the contents of an existing file to another file.

Parameters

Input

Name	Type	Description
sourceFileName	string	File to copy.
targetFileName	string	Copy of the original file.

Return Value

Name	Type	Description
result	integer	Result of the copy operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File not found or directory in path not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete operation.
`-5`	Invalid path for either of the filenames.
`-6`	Access denied.
`-7`	Specified path is a directory, not a file.
`-8`	Disk is full.

Parameter Discussion

sourceFileName and **targetFileName** can contain wildcard characters '?' and '*'. If **sourceFileName** has wildcards, `CopyFile` copies all matching files. If **targetFileName** has wildcards, the function matches it to **sourceFileName**. If the target file is a directory, `CopyFile` copies the existing file or group of files into the directory.

sourceFileName also can be the empty string `""`, in which case `CopyFile` copies the file found by the most recent call to `GetFirstFile` or `GetNextFile`.

CVILowLevelSupportDriverLoaded

```
int loaded = CVILowLevelSupportDriverLoaded (void);
```

Note ***Only the Windows 95/NT versions of LabWindows/CVI support*** `CVILowLevelSupportDriverLoaded`**.**

Purpose

Returns an indication of whether the LabWindows/CVI low-level support driver was loaded at startup. Table 8-2 lists Utility Library functions that require the LabWindows/CVI low-level driver.

Table 8-2. Functions That Require Low-Level Driver

Function	Platforms
`inp`	Windows NT
`inpw`	Windows NT
`outp`	Windows NT
`outpw`	Windows NT
`ReadFromPhysicalMemory`	Windows 95/NT
`ReadFromPhysicalMemoryEx`	Windows 95/NT
`WriteToPhysicalMemory`	Windows 95/NT
`WriteToPhysicalMemoryEx`	Windows 95/NT
`MapPhysicalMemory`	Windows 95/NT
`UnMapPhysicalMemory`	Windows 95/NT
`DisableInterrupts`	Windows 95
`EnableInterrupts`	Windows 95
`DisableTaskSwitching`	Windows 95

Note ***Most of these functions do not return an error if the low-level support driver is not loaded. To make sure your calls to these functions can execute correctly, call*** `CVILowLevelSupportDriverLoaded` ***at the beginning of your program.***

Both the LabWindows/CVI development environment and run-time engine automatically load the low-level support driver at startup if it is on disk. The low-level support driver ships with LabWindows/CVI. The Create Distribution Kit command in the Project window gives you an option to include it with your standalone executables or DLLs.

Parameters

None.

Return Value

Name	Type	Description
loaded	integer	Indicates whether the LabWindows/CVI low-level support driver was loaded at startup.

Return Codes

Code	Description
`1`	Low-level support driver was loaded at startup.
`0`	Low-level support driver was not loaded at startup.

CVIRTEHasBeenDetached

```
int hasBeenDetached = CVIRTEHasBeenDetached (void);
```

Note ***Only the Windows 95/NT versions of LabWindows/CVI support*** `CVIRTEHasBeenDetached`**.**

Purpose

Indicates whether Windows 95/NT has detached the LabWindows CVI Run-time Engine from your standalone executable process. The operating system detaches the run-time engine from a process in the following cases:

- The process terminates.
- The process dynamically unloads a DLL that uses the run-time engine, and the process does not directly link to the run-time engine.

You might need to use `CVIRTEHasBeenDetached` if you call LabWindows/CVI library functions in response to a `PROCESS_DETACH` message in the `DllMain` function of a DLL. In some cases, Windows 95/NT sends a `PROCESS_DETACH` message to the LabWindows/CVI Run-time Engine before it sends a `PROCESS_DETACH` message to your DLL. When the LabWindows/CVI Run-time Engine receives a `PROCESS_DETACH` message, it releases all the system resources it has acquired. When, in response to the `PROCESS_DETACH` message, your DLL calls LabWindows/CVI library functions that assume the system resources are still present, a general protection fault occurs.

A similar problem can occur when you call the `atexit` function in a DLL to register a routine for the ANSI C library to execute when your DLL unloads. The ANSI C library calls your routine when the DLL receives a `PROCESS_DETACH` message. This can occur after the LabWindows/CVI Run-time Engine receives a `PROCESS_DETACH` message. If your registered function calls LabWindows/CVI library functions that assume the system resources are still present, a general protection fault occurs.

To prevent such problems, call LabWindows/CVI functions from the `PROCESS_DETACH` code and registered functions in your DLL only if `CVIRTEHasBeenDetached` returns `0`.

Note `CVIRTEHasBeenDetached` ***always returns*** `0` ***when you call it in the LabWindows/CVI development environment or on platforms other than Windows 95/NT.***

Parameters

None.

Return Value

Name	Type	Description
hasBeenDetached	integer	`1` = The run-time engine has already received the `PROCESS_DETACH` message. Do not call LabWindows/CVI library functions. `0` = The run-time engine has not yet received a `PROCESS_DETACH` message. You can safely call LabWindows/CVI library functions.

Example:

```
int myPanel = 0;
static void CleanupPanels (void)
{
   if ( ! CVIRTEHasBeenDetached())
      if (myPanel > 0)
         DiscardPanel(myPanel);
}
int CreatePanel (void)
{
   if ((myPanel = LoadPanel (0,  “my.uir”,  MY_PANEL) < 0)
      return 0;
   atexit (CleanupPanels)
   return 1;
}
```

DateStr

```
char *s = DateStr (void);
```

Purpose

Returns a 10-character string in the form *MM-DD-YYYY*, where *MM* is the month, *DD* is the day, and *YYYY* is the year.

Parameters

None.

Return Value

Name	Type	Description
s	10-character string	Date in *MM-DD-YYYY* format.

Delay

```
void Delay (double numberOfSeconds);
```

Purpose

Waits the number of seconds **numberOfSeconds** indicates. The resolution under Windows is normally 1 ms. If, however, you set the `useDefaultTimer` configuration option to `True`, the resolution is 55 ms.

The resolution on Sun Solaris is 1 ms.

Parameter

Input

Name	Type	Description
numberOfSeconds	double-precision	Number of seconds to wait.

Return Value

None.

DeleteDir

```
int result = DeleteDir (char directoryName[]);
```

Purpose

Deletes an existing directory.

Parameter

Input

Name	Type	Description
directoryName	string	Name of directory to delete.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	Directory not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete operation.
`-6`	Access denied, or directory not empty.
`-7`	Path is a file, not a directory.

DeleteFile

```
int result = DeleteFile (char fileName[]);
```

Purpose

Deletes an existing file from disk.

Parameter

Input

Name	Type	Description
fileName	string	File to delete.

Return Value

Name	Type	Description
result	integer	Result of the delete operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File not found or directory in path not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete operation.
`-5`	Invalid path; for example, `c:filename` in Windows.
`-6`	Access denied.
`-7`	Specified path is a directory, not a file.

Parameter Discussion

fileName can contain wildcard characters '`?`' and '`*`', in which case `DeleteFile` deletes all matching files.

fileName also can be the empty string `""`, in which case `DeleteFile` deletes the file found by the most recent call to `GetFirstFile` or `GetNextFile`.

DisableBreakOnLibraryErrors

```
void DisableBreakOnLibraryErrors (void);
```

Purpose

`DisableBreakOnLibraryErrors` directs LabWindows/CVI not to display a run-time error dialog box when a National Instruments library function reports an error.

In general, you should use the Break on Library Errors checkbox in the **Run Options** command of the Project window to disable this option. However, you can use this function in conjunction with `EnableBreakOnLibraryErrors` to temporarily suppress the Break on Library Errors feature around a segment of code. `DisableBreakOnLibraryErrors` does not affect the state of the Break on Library Errors checkbox in the **Run Options** command of the Project window.

Note `SetBreakOnLibraryErrors` ***obsoletes*** `DisableBreakOnLibraryErrors`**.**

Parameters

None.

Return Value

None.

DisableInterrupts

```
void DisableInterrupts (void);
```

Purpose

Uses the CLI instruction to turn off all maskable 80x86 interrupts under Windows 3.1 and Windows 95. Under UNIX, `DisableInterrupts` uses `sigblock` to block all blockable signals.

Note ***Under Windows 95,*** `DisableInterrupts` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`***.***

Note ***Under Windows NT,*** `EnableInterrupts` ***and*** `DisableInterrupts` ***have no effect. Interrupts are always enabled while your program is running at the user level, as opposed to the kernel level.***

Parameters

None.

Return Value

None.

See Also

`CVILowLevelSupportDriverLoaded`

DisableTaskSwitching

```
void DisableTaskSwitching (void);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `DisableTaskSwitching`**.**

Purpose

Prevents the user from using one of the following Windows features to switch another task:

- The <Alt-Tab>, <Alt-Esc>, or <Ctrl-Esc> key combination under Windows 3.1 or Windows 95.
- The **Switch To** item in the system menu under Windows 3.1.

`DisableTaskSwitching` affects the behavior of these keys only while LabWindows/CVI or a LabWindows/CVI standalone executable is the active application under Windows.

If you configure Windows 95 to hide the taskbar, `DisableTaskSwitching` also prevents the user from using the mouse to bring the taskbar back up.

`DisableTaskSwitching` has no effect in Windows NT. Refer to the *Alternatives under Windows NT* section in this function description for instructions on how to achieve the desired effect.

Note ***Under Windows 95,*** `DisableTaskSwitching` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`**.**

Parameters

None.

Return Value

None.

See Also

`CVILowLevelSupportDriverLoaded`

Disabling the Task List

`DisableTaskSwitching` does not prevent the user from clicking on the desktop to get the task list under Windows 3.1 or from clicking on the taskbar under Windows 95. You can prevent the user from clicking on the desktop by forcing your window to cover the entire screen.

Forcing Window to Cover Entire Screen

You can force your window to cover the entire screen by making the following calls to functions in the User Interface Library.

```
SetPanelAttribute (panel, ATTR_SIZABLE, FALSE);
SetPanelAttribute (panel, ATTR_CAN_MINIMIZE, FALSE);
SetPanelAttribute (panel, ATTR_CAN_MAXIMIZE, FALSE);
SetPanelAttribute (panel, ATTR_SYSTEM_MENU_VISIBLE, FALSE);
SetPanelAttribute (panel, ATTR_MOVABLE, FALSE);
SetPanelAttribute (panel, ATTR_WINDOW_ZOOM, VAL_MAXIMIZE);
```

In these calls, `panel` is the panel handle for your top-level window. These calls work under Windows 3.1, Windows 95, and Windows NT.

Alternatives under Windows 3.1

Under Windows 3.1, you can prevent the user accessing the task list by disabling the Task Manager. You can do this by changing the following line in your `system.ini [boot]` section from:

```
taskman.exe = taskman.exe
```

to:

```
taskman.exe =
```

Forcing your window to cover the entire screen or disabling the Task Manager does not prevent the user from task switching using the <Alt-Tab> and <Alt-Esc> key combinations. You also must call `DisableTaskSwitching` to disable the <Alt-Tab> and <Alt-Esc> key combinations. As an alternative to calling `DisableTaskSwitching`, you can arrange for Windows to open your standalone application in place of the Program Manager when Windows boots. You can do this by changing the following line in your `system.ini [boot]` section from:

```
shell = progman.exe
```

to:

```
shell = full-path-of-your-executable
```

Alternatives under Windows 95

Under Windows 95, you can arrange for your standalone application to appear in place of the desktop when Windows boots.

You can do this by changing the following line in your `system.ini [boot]` section from:

```
shell = Explorer.exe
```

to:

```
shell = full-path-of-your-executable
```

Alternatives under Windows NT

Under Windows NT, you can achieve the same results as `DisableTaskSwitching` by arranging for your LabWindows/CVI application to be brought up in place of the Program Manager and by disabling the Task Manager. You can do this by making the following changes to the registry entry for the key name:

```
KEY_LOCAL_MACHINE\Software\Microsoft\Windows NT
\CurrentVersion\Winlogon
```

- Change the value for `shell` to the pathname of your application executable.
- Add a value with the name `taskman`. Set the data to an empty string.

Preventing Interference with Real-Time Processing

Under Windows, many user actions can interfere with real-time processing. The following actions suspend the processing of events:

- Moving and sizing top-level windows
- Bringing down the **System** menu
- Pressing the <Alt-Tab> key combination

You can prevent these user actions from interfering with event processing by:

- Calling `DisableTaskSwitching` or using the alternative this section mentions for Windows NT.
- Making all your top-level panels non-movable and non-sizable.
- Not using the Standard I/O Window in your final application.
- Making the following calls if you use any of the built-in pop-ups in the User Interface Library:

  ```
  SetSystemPopupsAttribute (ATTR_MOVABLE, 0);
  SetSystemPopupsAttribute (ATTR_SYSTEM_MENU_VISIBLE, 0);
  ```

An alternative approach is available on Windows 95/NT. You can enable timer control callbacks while the user presses <Alt-Tab>, pulls down the **System** menu, or, in some cases, moves or sizes a window. You can do this by using the following function call:

```
SetSystemAttribute (ATTR_ALLOW_UNSAFE_TIMER_EVENTS, 1);
```

This alternative approach is incomplete and can be unsafe. Refer to the discussion on *Unsafe Timer Events* in the *Using the System Attributes* section of Chapter 3, *Programming with the User Interface Library*, of the *LabWindows/CVI User Interface Reference Manual*.

See Also

`EnableTaskSwitching`

EnableBreakOnLibraryErrors

```
void EnableBreakOnLibraryErrors (void);
```

Purpose

`EnableBreakOnLibraryErrors` directs LabWindows/CVI to display a run-time error dialog box when a National Instruments library function reports an error. If you disable debugging, `EnableBreakOnLibraryErrors` has no effect.

In general, you should check the Break on Library Errors checkbox in the **Run Options** command of the Project window to enable this feature. However, you can use `EnableBreakOnLibraryErrors` in conjunction with `DisableBreakOnLibraryErrors` to temporarily suppress the Break on Library Errors feature around a segment of code. `EnableBreakOnLibraryErrors` does not affect the state of the Break on Library Errors checkbox.

Note `SetBreakOnLibraryErrors` ***obsoletes*** `EnableBreakOnLibraryErrors`**.**

Parameters

None.

Return Value

None.

EnableInterrupts

```
void EnableInterrupts (void);
```

Purpose

Uses the STI instruction to turn on all maskable 80x86 interrupts under Windows 3.1 and Windows 95. Under UNIX, `EnableInterrupts` reverses the effect of the last call to `DisableInterrupts`. It restores the signal processing state to the condition prior to the `DisableInterrupts` call.

Note ***Under Windows 95,*** `EnableInterrupts` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`***.***

Note ***Under Windows NT,*** `EnableInterrupts` ***and*** `DisableInterrupts` ***have no effect. Interrupts are always enabled while your program is running at the user level, as opposed to the kernel level.***

Parameters

None.

Return Value

None.

See Also

`CVILowLevelSupportDriverLoaded`

EnableTaskSwitching

```
void EnableTaskSwitching (void);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `EnableTaskSwitching`**.**

Purpose

Lets the user switch to another task by using the <Alt-Tab>, <Alt-Esc>, and <Ctrl-Esc> key combinations, and the **Switch To** item in the **Control/System** menu.
`EnableTaskSwitching` affects the behavior of these keys only while LabWindows/CVI or a LabWindows/CVI standalone executable is the active application.

Parameters

None.

Return Value

None.

See Also

`DisableTaskSwitching`

ExecutableHasTerminated

```
int status = ExecutableHasTerminated (int executableHandle);
```

Purpose

Determines whether an application you started with `LaunchExecutableEx` has terminated.

Parameter

Input

Name	Type	Description
executableHandle	integer	Executable handle you obtain from `LaunchExecutableEx`.

Return Value

Name	Type	Description
status	integer	Result of the operation.

Return Codes

Code	Description
`-1`	Handle is invalid.
`0`	Executable is still running.
`1`	Executable has been terminated.

Note ***If you launch a LabWindows/CVI executable under Windows 3.x, the launched executable process terminates itself after it launches a copy of the LabWindows/CVI Run-time Engine.*** `ExecutableHasTerminated` ***then always returns*** `1` ***because it cannot track the process identification for the second run-time engine. Refer to*** `LaunchExecutableEx` ***for more information.***

GetBreakOnLibraryErrors

```
int state = GetBreakOnLibraryErrors (void);
```

Purpose

Returns the state of the Break on Library Errors option. It returns a `1` if you enable the Break on Library Errors option. If you disable debugging, `GetBreakOnLibraryErrors` always returns `0`.

You can change the state of the Break on Library Errors option interactively using the **Run Options** command of the Project window. You can change the state of the Break on Library Errors option programmatically using `SetBreakOnLibraryErrors`.

Parameters

None.

Return Value

Name	Type	Description
state	integer	Current state of the Break on Library Errors option.

Return Codes

Code	Description
`1`	Break on Library Errors option is enabled.
`0`	Break on Library Errors option is disabled or debugging is disabled.

See Also

`SetBreakOnLibraryErrors`

GetBreakOnProtectionErrors

```
int state = GetBreakOnProtectionErrors (void);
```

Purpose

Returns the state of the Break on Protection Errors feature. It returns a `1` if you enable the option. If you disable debugging `GetBreakOnProtectionErrors` always returns `0`.

For more information, refer to `SetBreakOnProtectionErrors`.

Parameters

None.

Return Value

Name	Type	Description
state	integer	Current state of the Break on Protection Errors option.

Return Codes

Code	Description
`1`	Break on Protection Errors option is enabled.
`0`	Break on Protection Errors option is disabled or debugging is disabled.

See Also

`SetBreakOnProtectionErrors`

GetCurrentPlatform

```
int platformCode = GetCurrentPlatform (void);
```

Purpose

Returns a code that represents the operating system under which a project or standalone executable is running.

Do not confuse the return value of `GetCurrentPlatform` with the predefined macros such as `_NI_mswin_`, `_NI_unix_`, and others, which specify the platform on which you compile a source file.

`GetCurrentPlatform` is useful when you have a program that can run under multiple operating systems but must take different actions on the different systems. For example, consider a standalone executable that can run under Sun Solaris 2 and Windows 95/ NT. If the program must behave differently on the two platforms, you can use `GetCurrentPlatform` to determine the platform at run time.

Parameters

None.

Return Value

Name	Type	Description
platformCode	integer	Indicates the current operating system.

Return Codes

Message	Code	Platform
`kPlatformWin16`	1	Windows 3.1
`kPlatformWin95`	2	Windows 95
`kPlatformWinnt`	3	Windows NT
`kPlatformSunos4`	4	Sun Solaris 1
`kPlatformSunos5`	5	Sun Solaris 2
`kPlatformHPUX9`	6	HP-UX 9.*x*
`kPlatformHPUX10`	7	HP-UX 10.*x*

Note `GetCurrentPlatform` ***returns*** `kPlatformWin16` ***when you call it from a program that is running under Windows 95 but that was built using LabWindows/CVI for Windows 3.1.***

GetCVIVersion

```
int versionNum = GetCVIVersion (void);
```

Purpose

Returns the version of LabWindows/CVI you are running. In a standalone executable, this tells you which version of the LabWindows/CVI Run-time Libraries you are using.

The value is in the form `Nnn`, where the `N.nn` is the version number the About LabWindows/CVI dialog box shows.

For example, for LabWindows/CVI version 5.0, `GetCVIVersion` returns 500. For version 4.0.1, it returns 401. The values always increase with each new version of LabWindows/CVI.

Do not confuse the return value of `GetCVIVersion` with the predefined macro `_CVI_`, which specifies the version of LabWindows/CVI in which you compile the source file that contains the macro.

Parameters

None.

Return Value

Name	Type	Description
versionNum	integer	Version number of LabWindows/CVI or the Run-time Libraries.

Return Code

Code	Description
`N`*nn*	Where `N.`*nn* is the LabWindows/CVI version.

GetDir

```
int result = GetDir (char currentDirectory[]);
```

Purpose

Gets the current working directory on the default drive.

Parameter

Output

Name	Type	Description
currentDirectory	string	Current directory.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete operation.

Parameter Discussion

currentDirectory must be at least `MAX_PATHNAME_LEN` bytes long.

GetDrive

```
int result = GetDrive (int *currentDriveNumber, int *numberOfDrives);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `GetDrive`**.**

Purpose

Gets the current default drive number and the total number of logical drives in the system.

Parameters

Output

Name	Type	Description
currentDriveNumber	integer	Current default drive number.
numberOfDrives	integer	Number of logical drives.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	Current directory is on a network drive that is not mapped to a local drive. **currentDriveNumber** is set correctly, but **numberOfDrives** is set to `-1`.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete the operation.
`-6`	Access denied.

Parameter Discussion

The mapping between the drive number and the logical drive letter is 0 = A, 1 = B, and so on.

The total number of logical drives includes floppy-disk drives, hard-disk drives, RAM disks, and networked drives.

GetExternalModuleAddr

```
void *address = GetExternalModuleAddr (char name[], int moduleID,
                                       int *status);
```

Purpose

Obtains the address of an identifier in a module you loaded using `LoadExternalModule` or `LoadExternalModuleEx`.

Parameters

Input

Name	Type	Description
name	string	Name of the identifier.
moduleID	integer	ID of the loaded module.

Output

Name	Type	Description
status	integer	Zero or error code.

Return Value

Name	Type	Description
address	void pointer	Address of the identifier.

Return Codes

Code	Description
0	Success.
-1	Out of memory.
-4	Invalid file format.
-5	Undefined references.
-8	Cannot open file.
-9	Invalid module ID.
-10	Identifier not defined globally in the module.
-25	DLL initialization failed, for example, DLL file not found.

Parameter Discussion

moduleID is the value `LoadExternalModule` returns.

name is the name of the identifier, the address of which you obtain from the external module. The identifier must be a variable or function name defined globally in the external module.

status is zero if the function is a success or a negative error code if it fails.

If `GetExternalModuleAddr` succeeds, it returns the address of the variable or function in the module. If `GetExternalModuleAddr` fails, it returns `NULL`.

If the return value is the address of a function that has a calling convention different from the default calling convention, you must include the calling convention in the declaration of the function pointer. For example, if the function is declared in the external module as

```
int __stdcall SetADouble (double d);
```

and the default calling convention is `__cdecl`, you should declare the function pointer as

```
int (__stdcall * SetADouble_FnPtr)(double d) = NULL;.
```

Select **Options»Compiler Options** in the Project window to determine the default calling convention.

Example

```
void (*funcPtr) (char buf[], double dval, int *ival);
int module_id;
int status;
char buf[100];
double dval;
int ival;
char *pathname;
char *funcname;
pathname = "EXTMOD.OBJ";
funcname = "my_function";
module_id = LoadExternalModule (pathname);
if (module_id < 0)
   FmtOut ("Unable to load %s\n", pathname);
else
   {
   funcPtr = GetExternalModuleAddr (module_id, funcname, &status);
   if (funcPtr == NULL)
      FmtOut ("Could not get address of %s\n", funcname);
   else
      (*funcPtr) (buf, dval, &ival);
   }
```

GetFileAttrs

```
int result = GetFileAttrs (char fileName[], int *read-only, int *system,
                           int *hidden, int *archive);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `GetFileAttrs`**.**

Purpose

Gets the **read-only**, **system**, **hidden**, and **archive** attributes of a file.

The **read-only** attribute makes it impossible to write to the file or create a file with the same name.

The **system** attribute and **hidden** attribute both prevent the file from appearing in a directory list and exclude it from normal searches.

The operating system sets the **archive** attribute whenever you modify the file. The DOS `backup` command clears the **archive** attribute.

Parameters

Input

Name	Type	Description
fileName	string	File to get the attributes of.

Output

Name	Type	Description
read-only	integer	Read-only attribute.
system	integer	System attribute.
hidden	integer	Hidden attribute.
archive	integer	Archive attribute.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
0	Success.
1	Specified file is a directory.
-1	File not found.

Parameter Discussion

Each attribute parameter contains one of the following values:

0—attribute is not set

1—attribute is set

fileName can be the empty string "", in which case GetFileAttrs returns the attributes of the file that the most recent call to GetFirstFile or GetNextFile found.

Example

```
/* Get the attributes of WAVEFORM.DAT. */
int read_only, system, hidden, archive;
GetFileAttrs ("WAVEFORM.DAT",&read_only,&system,&hidden,&archive);
if (read_only)
   FmtOut("WAVEFORM.DAT is a read-only file!");
```

GetFileDate

```
int result = GetFileDate (char fileName[], int *month, int *day, int *year);
```

Purpose

Gets the date of a file.

Parameters

Input

Name	Type	Description
fileName	string	File to get the date of.

Output

Name	Type	Description
month	integer	Month; 1–12.
day	integer	Day of month; 1–31.
year	integer	Year; 1980–2099.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File not found or directory in path not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete operation.
`-5`	Invalid path, for example, `c:filename` in Windows.
`-6`	Access denied.

Parameter Discussion

fileName can be the empty string `""`, in which case `GetFileDate` returns the date of the file that the most recent call to `GetFirstFile` or `GetNextFile` found.

Example

```
/* Get the date of WAVEFORM.DAT. */
int month, day, year;
GetFileDate ("WAVEFORM.DAT", &month, &day, &year);
```

GetFileSize

```
int result = GetFileSize (char fileName[], long *fileSize);
```

Purpose

Returns the size of a file.

Parameters

Input

Name	Type	Description
fileName	string	Name of the file.

Output

Name	Type	Description
fileSize	long	Size of the file in bytes.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File not found or directory in path not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete the operation.
`-5`	Invalid path; for example, `c:filename` in Windows.
`-6`	Access denied.

Parameter Discussion

fileName can be the empty string `""`, in which case `GetFileSize` returns the size of the file that the most recent call to `GetFirstFile` or `GetNextFile` found.

Example

```
/* Get the size of WAVEFORM.DAT. */
long size;
if (GetFileSize ("WAVEFORM.DAT",&size) == 0)
   FmtOut("The size of WAVEFORM.DAT is %i[b4]",size);
```

GetFileTime

```
int result = GetFileTime (char fileName[], int *hours, int *minutes,
                          int *seconds);
```

Purpose

Gets the time of a file.

Parameters

Input

Name	Type	Description
fileName	string	File to get the date of.

Output

Name	Type	Description
hours	integer	Hours; 0–23.
minutes	integer	Minutes; 0–59.
seconds	integer	Seconds; 0–58, odd values are rounded down.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File not found or directory in path not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete the operation.
`-5`	Invalid path; for example, `c:filename` in Windows.
`-6`	Access denied.

Parameter Discussion

fileName can be the empty string `""`, in which case `GetFileTime` returns the time of the file the most recent call to `GetFirstFile` or `GetNextFile` found.

Example

```
/* Get the time of WAVEFORM.DAT. */
int hours, minutes, seconds;
GetFileTime ("WAVEFORM.DAT", &hours, &minutes, &seconds);
```

GetFirstFile

```
int result = GetFirstFile (char searchPath[], int normal, int read-only,
                           int system, int hidden, int archive,
                           int directory, char fileName[]);
```

Purpose

Conducts a search for files with specified attributes and returns the first matching file. Call `GetNextFile` to get the names of other matching files.

If you select multiple attributes, a match occurs on the first file for which one or more of the specified attributes are set and that matches the pattern in **searchPath**. The search attributes are **normal**, **read-only**, **system**, **hidden**, **archive**, and **directory**.

Under UNIX, `GetFirstFile` honors only the **directory** attribute. If you pass `1` for the **directory** attribute, only directories match. If you pass `0` for the **directory** attribute, only non-directories match.

Under Windows, `GetFirstFile` honors all the attributes. The **normal** attribute specifies files with no attributes set or with only the archive bit set. The **archive** attribute specifies files that have been modified because they were last backed up using the DOS `backup` command. The **read-only** attribute specifies files that are protected from modification or overwriting. The **system** and **hidden** attributes specify files that normally do not appear in a directory listing. The **directory** attribute specifies directories.

If you use only the **normal** attribute, `GetFirstFile` can return any file that is not read only, not a system file, not hidden, and not a directory. Normal files can have the **archive** attribute on or off.

If you specify the **read-only** attribute, `GetFirstFile` can return any file that is read only unless the file is a system, or hidden, file and you did not specify the **system**, or **hidden** attribute.

If you specify the **system** attribute, `GetFirstFile` can return any system file unless the file is also a hidden file and you did not specify the **hidden** attribute. If you do not specify the **system** attribute, a system file cannot match regardless of its other attributes.

If you specify the **hidden** attribute, `GetFirstFile` can return any hidden file unless the file is also a system file and you did not specify the **system** attribute. If you do not specify the **hidden** attribute, a hidden file cannot match regardless of its other attributes.

If you use more than one attribute, the effect is additive. `GetFirstFile` returns any file that meets only one of the attributes you specify regardless of the additional attributes you specify.

Parameters

Input

Name	Type	Description
searchPath	string	Path to search.
normal	integer	Normal attribute.
read-only	integer	Read-only attribute.
system	integer	System attribute.
hidden	integer	Hidden attribute.
archive	integer	Archive attribute.
directory	integer	Directory attribute.

Output

Name	Type	Description
fileName	string	First file found.

Return Value

Name	Type	Description
result	integer	Result of search.

Return Codes

Code	Description
`0`	Success.
`-1`	No files found that match criteria.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete the operation.
`-5`	Invalid path, for example, `c:filename` in Windows.
`-6`	Access denied.

Parameter Discussion

searchPath can contain the wildcard characters '*' and '?'.

Each attribute parameter can have one of the following values:

`0`— do not search for files with the attribute

`1`— search for files with the attribute

fileName contains the basename and extension of the first matching file and must be at least `MAX_FILENAME_LEN` characters in length.

GetFullPathFromProject

```
int result = GetFullPathFromProject (char fileName[], char fullPathName[]);
```

Purpose

Gets the full pathname for the file you specify, if the file is in the currently loaded project.

Parameters

Input

Name	Type	Description
fileName	string	Name of the file in the project.

Output

Name	Type	Description
fullPathName	string	Full pathname of the file.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File is not in the project.

Parameter Discussion

fileName is the name of a file that is in the currently loaded project. The name must be a simple filename and should not contain any directory paths. For example, `file.c` is a simple filename, whereas `dir\file.c` is not.

fullPathName must be at least `MAX_PATHNAME_LEN` bytes long.

Using This Function

`GetFullPathFromProject` is useful when your program needs to access a file in the project and you do not know what directory the file is in.

Example

```
char *fileName;
char fullPath[MAX_PATHNAME_LEN];
fileName = "myfile.c";
if (GetFullPathFromProject (fileName, fullPath) < 0)
   FmtOut ("File %s is not in the project\n", fileName);
```

Note ***LabWindows/CVI does not report run-time errors for*** `GetFullPathFromProject`.

GetInterruptState

```
int interruptstate = GetInterruptState (void);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `GetInterruptState`**.**

Purpose

Returns the state of the interrupt bit of the 80x86 CPU status flag.

Under Windows NT, `GetInterruptState` always returns `1`. Interrupts are always enabled while your program is running at the user level, as opposed to the kernel level.

Parameters

None.

Return Value

Name	Type	Description
interruptstate	integer	Interrupt bit of 80x86 CPU status flag.

GetKey

```
int k = GetKey (void);
```

Purpose

Waits for the user to press a key and returns the key code as an integer value.

Note `GetKey` ***detects keystrokes only in the Standard I/O window. It does not detect keystrokes in windows you create with the User Interface Library or in the console window in a Windows Console Application.***

Parameters

None.

Return Value

Name	Type	Description
k	integer	Key code.

Using This Function

The values `GetKey` returns are the same as the key values the User Interface Library uses. Refer to `userint.h`. Table 8-3 shows examples of keystrokes and the values `GetKey` returns for them.

Table 8-3. Example Keystrokes and GetKey Return Values

Keystroke	Return Value
<b>	`'b'`
<Ctrl-b>	`(VAL_MENUKEY_MODIFIER | 'B')`
<F4>	`VAL_F4_VKEY`
<Shift-F4>	`(VAL_SHIFT_MODIFIER | VAL_F4_VKEY)`

Note `GetKey` ***returns*** `-1` ***if you are running under UNIX and have done one of the following:***

- ***Selected*** **Options»Environment»Use hosts system's Standard Input/Output** ***in the Project window***
- ***Called*** `SetStdioPort` ***to set the port to*** `HOST_SYSTEM_STDIO`

Example

```
/* Give the user a chance to quit the program. */
int k;
FmtOut ("Enter 'q' to quit, any other key to continue");
k = GetKey ();
if ((k == 0x0051) || (k == 0x0071))   /* q or Q */
       exit (0);
```

GetModuleDir

```
int result = GetModuleDir (void *moduleHandle, char directoryPathname[]);
```

Note ***Only the Windows 95/NT versions of LabWindows/CVI support*** `GetModuleDir`.

Purpose

Obtains the name of the directory of the DLL module you specify.

`GetModuleDir` is useful when you distribute a DLL and its related files to multiple users who might place them in different directories. If your DLL needs to access a file that is in the same directory as the DLL, you can use `GetModuleDir` and `MakePathname` to construct the full pathname.

If the moduleHandle you specify is zero, `GetModuleDir` returns the same result as `GetProjectDir`.

Parameters

Input

Name	Type	Description
moduleHandle	void pointer	Module handle of DLL, or zero for the project.

Output

Name	Type	Description
directoryPathname	string	Directory of the module.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
0	Success.
-1	Current project has no pathname; that is, it is untitled.
-2	There is no current project.
-3	Out of memory.
-4	Operating system is unable to determine the module directory; **moduleHandle** is probably invalid.

Parameter Discussion

directoryPathname must be at least `MAX_PATHNAME_LEN` bytes long.

If you want to obtain the directory name of the DLL in which the call to `GetModuleDir` resides, then pass `__CVIUserHInst` as the **moduleHandle.** You can pass any valid Windows module handle. If you pass `0` for the **moduleHandle**, `GetModuleDir` obtains the directory of the project or standalone executable.

GetNextFile

```
int result = GetNextFile (char fileName[]);
```

Purpose

Gets the next file found in the search that `GetFirstFile` starts.

Parameter

Output

Name	Type	Description
fileName	string	Next file found.

Return Value

Name	Type	Description
result	integer	Result of the search.

Return Codes

Code	Description
`0`	Success.
`-1`	No more files found that match criteria.
`-2`	`GetFirstFile` must initiate search.

Parameter Discussion

fileName contains the basename and extension of the next matching file and must be at least `MAX_FILENAME_LEN` characters in length.

GetPersistentVariable

```
void GetPersistentVariable (int *value);
```

Purpose

Returns the value `SetPersistentVariable` sets. However, if you unloaded the project since you last called `SetPersistentVariable`, the function returns zero.

In a standalone executable, `GetPersistentVariable` returns zero if you have not called `SetPersistentVariable` since the start of execution.

Parameter

Output

Name	Type	Description
value	integer	Current value of the persistent variable.

Return Value

None.

GetProjectDir

```
int result = GetProjectDir (char directoryName[]);
```

Purpose

Gets the name of the directory that contains the currently loaded project file.

Parameter

Output

Name	Type	Description
directoryName	string	Directory of project.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	Current project has no pathname; it is untitled.

Parameter Discussion

directoryName must be at least `MAX_PATHNAME_LEN` bytes long.

Using This Function

`GetProjectDir` is useful when you distribute a project and its related files to multiple users who might place them in a different directory on each computer. If your program needs to access a file that is in the same directory as the project, you can use `GetProjectDir` and `MakePathname` to construct the full pathname.

Example

```
/* Get the name of the directory that contains myfile. */
char *fileName;
char projectDir[MAX_PATHNAME_LEN];
char fullPath[MAX_PATHNAME_LEN];
fileName = "myfile";
if (GetProjectDir (projectDir) < 0)
   FmtOut ("Project is untitled\n");
else
   MakePathname (projectDir, fileName, fullPath);
```

GetStdioPort

```
void GetStdioPort (int *stdioPort);
```

Note ***Only the UNIX versions of LabWindows/CVI support*** `GetStdioPort`**.**

Purpose

Gets a value that indicates the current destination for data you write to the Standard Output and the source of data you read from the Standard Input.

The Standard I/O port can be either the LabWindows/CVI Standard Input/Output window or the Standard Input/Output of the host system.

Parameter

Output

Name	Type	Description
stdioPort	integer	`0` = LabWindows/CVI Standard Input/Output window `1` = Standard Input/Output of host system

Return Value

None.

GetStdioWindowOptions

```
void GetStdioWindowOptions (int *maxNumLines, int *bringToFrontWhenModified,
                            int *showLineNumbers);
```

Purpose

Gets the current value of the following Standard Input/Output window options:

- Maximum number of lines
- Bring to front when modified
- Show line numbers

Parameters

Output

Name	Type	Description
maxNumLines	integer	Maximum number of lines you can store in the Standard Input/Output window. If this amount is exceeded, lines are discarded from the top.
bringToFrontWhenModified	integer	Indicates whether to bring the Standard Input/Output window to the front each time you add a string or character. `1` = Yes `0` = No
showLineNumbers	integer	Indicates whether to show line numbers in the Standard Input/Output window. `1` = Yes `0` = No

Return Value

None.

Parameter Discussion

If you do not want to obtain any of these values, you can pass `NULL`.

GetStdioWindowPosition

```
void GetStdioWindowPosition (int *top, int *left);
```

Purpose

Gets the current position, in pixels, of the client area of the Standard Input/Output window relative to the upper left corner of the screen. The client area begins under the title bar and to the right of the frame.

Parameters

Output

Name	Type	Description
top	integer	Current distance, in pixels, from the top of client area of the Standard Input/Output window to the top of the screen.
left	integer	Current distance, in pixels, from the left edge of the client area of the Standard Input/Output window to the left edge of the screen.

Return Value

None.

GetStdioWindowSize

```
void GetStdioWindowSize (int *height, int *width);
```

Purpose

Gets the height and width, in pixels, of the client area of the Standard Input/Output window. The client area excludes the frame and the title bar.

Parameters

Output

Name	Type	Description
height	integer	Current height, in pixels, of the client area of the Standard Input/Output window.
width	integer	Current width, in pixels, of the client area of the Standard Input/Output window.

Return Value

None.

GetStdioWindowVisibility

```
void GetStdioWindowVisibility (int *visible);
```

Purpose

Indicates whether the Standard Input/Output window is currently visible. If the window is minimized into an icon, `GetStdioWindowVisibility` considers the window to be *not* visible. If the you cannot see the window merely because its position is off the screen, `GetStdioWindowVisibility` considers the window to be visible.

Parameters

Output

Name	Type	Description
visible	integer	`1` = Standard I/O window is visible `0` = Standard I/O window is not visible

Return Value

None.

GetSystemDate

```
int status = GetSystemDate (int *month, int *day, int *year);
```

☞ **Note** ***Only the Windows versions of LabWindows/CVI support*** `GetSystemDate`**.**

Purpose

Obtains the system date in numeric format.

Parameters

Output

Name	Type	Description
month	integer	Month; 1–12.
day	integer	Day of the month; 1–31.
year	integer	Year; under Windows 3.1, the year is limited to the values 1980–2099.

Return Value

Name	Type	Description
status	integer	Success or failure.

Return Codes

Code	Description
`0`	Success.
`-1`	Operating system reported failure.

GetSystemTime

```
int status = GetSystemTime(int *hours, int *minutes, int *seconds);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `GetSystemTime`**.**

Purpose

Obtains the system time in numeric format.

Parameters

Output

Name	Type	Description
hours	integer	Hours; 0–23.
minutes	integer	Minutes; 0–59.
seconds	integer	Seconds; 0–59.

Return Value

Name	Type	Description
status	integer	Success or failure.

Return Codes

Code	Description
`0`	Success.
`-1`	Operating system reported failure.

GetWindowDisplaySetting

```
void GetWindowDisplaySetting (int *visible, int *zoomState);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `GetWindowDisplaySetting`.

Purpose

Indicates how the user of your application wants the initial application window to display. The values `GetWindowDisplaySetting` returns reflect the display options the user sets for the program in Program Manager and other Windows shells.

Parameters

Output

Name	Type	Description
visible	integer	`0` = hide window `1` = display window

zoomState	integer	`ATTR_NO_ZOOM` = normal display
		`ATTR_MINIMIZE`
		`ATTR_MAXIMIZE`

Return Value

None.

Example

If you want to honor the user's display options, put the following code where you display your initial panel:

```
int showWindow, zoomState;
GetWindowDisplaySetting (&showWindow, &zoomState);
/* Load panel or create panel. */
if (showWindow){
   SetPanelAttribute (panel, ATTR_WINDOW_ZOOM, zoomState);
   SetPanelAttribute (panel, ATTR_VISIBLE, 1);
}
```

InitCVIRTE

```
int status = InitCVIRTE (void *hInstance, char *argv[], void *reserved);
```

Purpose

Performs initialization of the LabWindows/CVI Run-time Engine. You need `InitCVIRTE` only in executables or DLLs that you link using an external compiler. Otherwise, the function is harmless.

Parameters

Input

Name	Type	Description
hInstance	void pointer	`0` if called from `main`. **hInstance** if called from `WinMain`, first parameter. **hInstDLL** if called from `DllMain`, first parameter.
argv	string array	**argv** if called from `main`, second parameter. Otherwise, `0`.
reserved	void pointer	Reserved for future use. Pass `0`.

Return Value

Name	Type	Description
status	integer	`1` = success `0` = failure, probably out of memory

Using this Function

You should call `InitCVIRTE` in your `main`, `WinMain`, or `DllMain` function. Which of these three functions you use determines the parameter values you pass to `InitCVIRTE`. The following examples show how to use `InitCVIRTE` in each case:

```
int main (int argc, char *argv[])
{
   if (InitCVIRTE (0, argv, 0) == 0)
      return -1; /* out of memory   */
   /* your other code */
   return 0;
}
int __stdcall WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
                        LPSTR lpszCmdLine, int nCmdShow)
{
   if (InitCVIRTE (hInstance, 0, 0) == 0)
      return -1; /* out of memory   */
   /* your other code */
   return 0;
}
int __stdcall DllMain (void *hinstDLL, int fdwReason, void
                        *lpvReserved)
{
   if (fdwReason == DLL_PROCESS_ATTACH)
      {
      if (InitCVIRTE (hinstDLL, 0, 0) == 0)
         return 0; /* out of memory */
         /* your other ATTACH code */
         }
   else if (fdwReason == DLL_PROCESS_DETACH)
      {
      /* your other DETACH code */
      CloseCVIRTE ();
      }
   return 1;
}
```

☞ **Note** ***The prototypes for*** `InitCVIRTE` ***and*** `CloseCVIRTE` ***are in*** `cvirte.h`***, which is included by*** `utility.h`**.**

inp

```
char byteRead = inp (int portNumber);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `inp`**.**

Purpose

Reads a byte from a port.

Note ***Under Windows NT,*** `inp` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`**.**

Parameter

Input

Name	Type	Description
portNumber	integer	Port

Return Value

Name	Type	Description
byteRead	char	Byte read from the port.

See Also

`CVILowLevelSupportDriverLoaded`

inpw

```
short wordRead = inpw (int portNumber);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `inpw`***.***

Purpose

Reads a word from a port.

Note ***Under Windows NT,*** `inpw` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`***.***

Parameter

Input

Name	Type	Description
portNumber	integer	Port.

Return Value

Name	Type	Description
wordRead	short	Word read from the port.

See Also

`CVILowLevelSupportDriverLoaded`

InStandaloneExecutable

```
int standalone = InStandaloneExecutable(void);
```

Purpose

Returns a nonzero value if your program is currently running as a standalone executable. If your program is currently running under the LabWindows/CVI development environment, `InStandaloneExecutable` returns zero.

Parameters

None.

Return Value

Name	Type	Description
standalone	integer	`1` = program is currently running as a standalone executable `0` = program is currently running under LabWindows/CVI

KeyHit

```
int result = KeyHit (void);
```

Purpose

Indicates whether the user has pressed a key on the keyboard.

Note `KeyHit` ***detects keystrokes only in the Standard I/O window. It does not detect keystrokes in windows you create with the User Interface Library or in the console window in a Windows console application.***

Parameters

None.

Return Value

Name	Type	Description
result	integer	Indicates if a key has been pressed.

Return Codes

Code	Description
`0`	Key has not been pressed.
`1`	Key has been pressed.

Using This Function

`KeyHit` returns `1` if a keystroke is available in the keyboard buffer; `0` otherwise. After a keystroke is available, you should make a call to `GetKey` to flush the keyboard buffer. Otherwise, `KeyHit` continues to return `1`.

Note `KeyHit` ***always returns*** `0` ***if you are running under UNIX and have done one of the following:***

- ***Selected*** **Options»Environment»Use Host System's Standard Input/Output** ***in the Project window***
- ***Called*** `SetStdioPort` ***to set the port to*** `HOST_SYSTEM_STDIO`

Example

```
/* Flush any pending keystrokes. */
while (KeyHit())
   GetKey();
/* Perform loop indefinitely until the user presses key. */
while (!KeyHit()) {
}
```

LaunchExecutable

```
int result = LaunchExecutable (char fileName[]);
```

Purpose

Starts running a program and returns without waiting for it to exit. The program must be an actual executable; that is, you cannot launch commands intrinsic to a command interpreter.

Under Windows the executable can be either a DOS or Windows executable, including `*.exe`, `*.com`, `*.bat`, and `*.pif` files.

If you need to execute a command built into `command.com` such as `copy`, `dir`, and others, you can call `LaunchExecutable` with the command

```
command.com /C DosCommand args
```

where `DosCommand` is the shell command you want to execute. For example, the following command string copies `file.tmp` from the `temp` directory to the `tmp` directory:

```
command.com /C copy c:\\temp\\file.tmp c:\\tmp
```

Note ***If you want to monitor whether the launched executable has terminated, use*** `LaunchExecutableEx`**.**

Parameter

Input

Name	Type	Description
fileName	string	Pathname of executable file and arguments.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes under UNIX

Code	Description
0	Command successfully started.
-1	Launching the executable would exceed the operating system limit on the total number of processes under execution or the total number of processes per user.
-2	Insufficient swap space for the new process.
-3	vfork failed for unknown reason.
-4	Search permission is denied for a directory listed in the path prefix of the new process image file, the new process image file denies execution permission, or the new process image file is not a regular file.
-5	The length of the pathname or filename or an element of the environment variable PATH prefixed to a filename exceeds PATH_MAX, or a pathname component is longer than NAME_MAX while _POSIX_NO_TRUNC is in effect for that file. Refer to the man page for pathconf(2V).
-6	One or more components of the pathname of the new process image file do not exist.
-7	A component of the path prefix of the new process image file is not a directory.
-8	Number of bytes that the new process-image-argument list and the environment list use is greater than ARG_MAX bytes. Refer to the man page for sysconf(2V).
-9	New process image file has the appropriate access permission but is not in the proper format.

Return Codes under Windows

Code	Description
0	Command was successfully started.
-1	System was out of memory, executable file was corrupt, or relocations were invalid.
-3	File was not found.
-4	Path was not found.

Code	Description
-6	Attempt was made to dynamically link to a task, or there was a sharing or network-protection error.
-7	Library required separate data segments for each task.
-9	There was insufficient memory to start the application.
-11	Windows version was incorrect.
-12	Executable file was invalid. Either it was not a Windows application or there was an error in the `.exe` image.
-13	Application was designed for a different operating system.
-14	Application was designed for MS-DOS 4.0.
-15	Type of executable file was unknown.
-16	You made an attempt to load a real-mode application developed for an earlier version of Windows.
-17	You made an attempt to load a second instance of an executable file that contains multiple data segments that were not marked read only.
-20	Attempt was made to load a compressed executable file. You must decompress the file before you can load it.
-21	DLL file was invalid. One of the DLLs required to run this application was corrupt.
-22	Application requires Windows 32-bit extensions.

Parameter Discussion

fileName is the program to run.

If the program is not in one of the directories specified in the `PATH` environment variable, you must specify the full path. The path can include arguments to pass to the program.

Under Windows, if the program is a `.pif`, `.bat`, or `.com` file, you must include the extension in the pathname. For example, under Windows, the following command string launches the Edit program with the file `file.dat`:

```
c:\\dos\\edit.com c:\\file.dat
```

See Also

`LaunchExecutableEx`

LaunchExecutableEx

```
int result = LaunchExecutableEx (char *fileName, int windowState,
                                 int *handle);
```

Purpose

Performs the same operation as `LaunchExecutable` with the following extended features:

- Under Windows, you can specify how the Windows application displays.
- `LaunchExecutableEx` returns a handle to the executable that can show whether the executable is still running and also can terminate the executable.

Parameters

Input

Name	Type	Description
fileName	string	Pathname of executable file and arguments.
windowState	integer	Specifies how to show a Windows program. Ignored under UNIX.

Output

Name	Type	Description
handle	integer	Handle that represents the executable launched.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
(nonzero value)	Failure, refer to `LaunchExecutable`.

Parameter Discussion

Table 8-4 shows valid values for **windowState**.

Table 8-4. Valid windowState Values

Value	Application window behavior
`LE_HIDE`	hidden
`LE_SHOWNORMAL`	shown normally and activated
`LE_SHOWMINIMIZED`	displayed as an icon and activated
`LE_SHOWMAXIMIZED`	displayed as a maximized window and activated
`LE_SHOWNA`	shown normally but not activated
`LE_SHOWMINNOACTIVE`	shown as an icon but not activated

You can pass the handle you obtain from `LaunchExecutableEx` to `ExecutableHasTerminated` and `TerminateExecutable`. When you no longer need the handle, you must call `RetireExecutableHandle`. If you do not want to obtain a handle, you can pass `NULL` for the **handle** parameter.

When you launch several processes with `LaunchExecutableEx` but do not call `RetireExecutableHandle` on them, you might reach the limit for the maximum number of processes the system imposes. This happens even if the processes terminate; the program does not recognize that the processes have terminated until you call `RetireExecutableHandle`.

Checking Termination of LabWindows/CVI Executables under Windows 3.1

If you launch another LabWindows/CVI executable under Windows 3.1, the launched executable process terminates itself after launching a copy of the LabWindows/CVI Run-time Engine. `ExecutableHasTerminated` then always returns `1` because it cannot track the process identification for the second run-time engine. This behavior can also occur with non-LabWindows/CVI executables.

You can work around this problem when launching LabWindows/CVI runtime executables by executing the run-time engine directly and passing it the pathname of the executable, as shown in the following example:

```
c:\cvi\cvirt5.exe  c:\test\myapp.exe
```

The pathname of the run-time engine might not be `c:\cvi\cvirt5.exe`. You can determine the pathname of the run-time engine by looking at the `cvirt5` section in `win.ini`. If the run-time executable was made with a different version of CVI, look in the `cvirt`*nn* section for that version.

If you need to pass arguments to your application, create a file that contains the arguments and pass the pathname of that file as the second argument to the run-time engine, as shown in the following example:

```
c:\cvi\cvirt5.exe  c:\test\myapp.exe  myargs
```

The file that contains the arguments must be in the same directory as the executable. The first three characters in the file that contains the arguments must be "CVI" in uppercase, as in the following example:

```
CVI  arg1  arg2  arg3
```

The run-time engine deletes the file that contains the arguments after reading it.

See Also

ExecutableHasTerminated, TerminateExecutable, RetireExecutableHandle

LoadExternalModule

```
int module_id = LoadExternalModule (char pathName[]);
```

Purpose

Loads a file that contains one or more object modules.

Parameter

Input

Name	Type	Description
pathName	string	Relative or absolute pathname of the module to load.

Return Value

Name	Type	Description
module_id	integer	ID of the loaded module.

Return Codes

Code	Description
`-1`	Out of memory.
`-2`	File not found.
`-4`	Invalid file format.
`-6`	Invalid pathname.
`-7`	Unknown file extension.
`-8`	Cannot open file.
`-11`	`.pth` file open error.
`-12`	`.pth` file read error.
`-13`	`.pth` file invalid contents.
`-14`	DLL header file contains a static function prototype.
`-15`	DLL function has an unsupported argument type.
`-16`	DLL has a variable argument function.
`-17`	DLL header contains a function without a proper prototype.

Code	Description
-18	DLL function has an unsupported return type.
-19	DLL function argument or return type is a function pointer.
-20	Function in the DLL header file was not found in the DLL.
-21	Could not load the DLL.
-22	Could not find the DLL header file.
-23	Could not load the DLL header file; out of memory or the file is corrupted.
-24	Syntax error in the DLL header file.
-25	DLL initialization function failed.
-26	Module already loaded with different calling module handle. Refer to `LoadExternalModuleEx`.
-27	Invalid calling module handle. Refer to `LoadExternalModuleEx`.
-28	Module you loaded in Borland mode within the LabWindows/CVI development environment contains uninitialized global variables that are also defined in other modules.

Parameter Discussion

`LoadExternalModule` loads an external object module file. You do not need to list the file in your project or load it as an instrument module.

Under Windows 3.1, the file can be an object file (`.obj`), a library file (`.lib`), or a dynamic link library (`.dll`). You must compile object and library modules with the Watcom C compiler for Windows or with the LabWindows/CVI compiler.

Under Windows 95/NT, the file can be an object file (`.obj`), a library file (`.lib`), or a DLL import library (`.lib`). You cannot load a DLL directly. You can compile object and library modules with LabWindows/CVI or with an external compiler.

Under UNIX, the file can be an object file (`.o`) or a statically linked library (`.a`).

All files must conform to the rules for loadable compiled modules in the *LabWindows/CVI Programmer Reference Manual*.

By loading external object modules, you can execute code that is not in your project and not in a loaded instrument module. You can load the external modules only when you need them and unload them when you no longer need them.

After you load a module, you can execute its code in one of two ways:

- You can obtain pointers to functions in the module by calling `GetExternalModuleAddr`. Then, you can call the module functions through the function pointers.
- You can call `RunExternalModule`. This requires that the module contain a function with a pre-defined name and prototype. The function serves as the entry point to the module. Refer to `RunExternalModule` for more information.

You can use `LoadExternalModule` on a source file (`.c`) that is part of the current project or on a source file that you load as the program for an instrument module. This allows you to develop your module in source code form and test it using the LabWindows/CVI debugging capabilities. After you finish testing your module and compile it into an external object or library file, you must change the pathname in the call to `LoadExternalModule` in your application source code. You do not have to make any other modifications to load the module.

Avoid calling `LoadExternalModule` on a file in the project when you plan to link your program in an external compiler. The LabWindows/CVI Utility Library does not know the locations of symbols in executables or DLLs linked in external compilers. You can provide this information by using the Other Symbols section of the **External Compiler Support** dialog box in the **Build** menu of the LabWindows/CVI Project window to create an object module that contains a table of symbols you want to find using `GetExternalModuleAddr`. If you use this method, you should pass the empty string `""` to `LoadExternalModule` for the module pathname.

If successful, `LoadExternalModule` returns an integer module ID that you can later pass to `RunExternalModule`, `GetExternalModuleAddr`, and `UnloadExternalModule`. If unsuccessful, `LoadExternalModule` returns a negative error code.

Resolving External References from Object and Static Library Files on Windows 95/NT

An important difference exists between loading an object or static library module and loading a DLL through an import library. DLLs are prelinked; that is, when you load a DLL, the loader does not need to resolve any external references. Object and static library modules, on the other hand, have unresolved external references. `LoadExternalModule` resolves them using symbols defined in the project or exported by object, static library, or import library modules that you have already loaded using `LoadExternalModule`. This is true even when you call `LoadExternalModule` in a DLL. `LoadExternalModule` does not use symbols in a DLL to resolve external references unless the DLL exports those symbols.

When you load an object or library module from a DLL, you might want to resolve external references in the object or library module through global symbols the DLL does not export. If you want to do this, you must call `LoadExternalModuleEx` rather than `LoadExternalModule`.

Using This Function

pathname can be a relative or absolute pathname. If it is a simple filename, such as `module.obj`, `LoadExternalModule` takes the following steps to find the file:

1. It first looks for the file in the project list.
2. It then looks for the file in the directory that contains the currently loaded project.
3. Under Windows 3.1, if the file is not found and its extension is `.dll`, `LoadExternalModule` searches for the file in the directories Windows searches to find DLLs.

If it is a relative pathname with one or more directory paths (such as `dir\module.obj`), `LoadExternalModule` creates an absolute pathname by appending the relative pathname to the directory that contains the currently loaded project.

If the **pathname** is for a DLL import library, `LoadExternalModule` finds the DLL using the DLL name embedded in the import library and the standard Windows DLL search algorithm.

See Also

`LoadExternalModuleEx`, `GetExternalModuleAddr`, `RunExternalModule`, `UnloadExternalModule`, `ReleaseExternalModule`

Example

```
void (*funcPtr) (char buf[], double dval, int *ival);
int module_id;
int status;
char buf[100];
double dval;
int ival;
char *pathname;
char *funcname;
pathname = "EXTMOD.OBJ";
funcname = "my_function";
module_id = LoadExternalModule (pathname);
if (module_id < 0)
   FmtOut ("Unable to load %s\n", pathname);
else {
       funcPtr = GetExternalModuleAddr (module_id, funcname, &status);
       if (funcPtr == NULL)
          FmtOut ("Could not get address of %s\n", funcname);
       else
          (*funcPtr) (buf, dval, &ival);
}
```

LoadExternalModuleEx

```
int moduleId = LoadExternalModuleEx (char pathName[],
                                     void *callingModuleHandle);
```

Purpose

Loads a file that contains one or more object modules. It is similar to `LoadExternalModule` except that under Windows 95/NT, you can resolve external references in object and library modules you load in a DLL by using symbols the DLL does not export. On platforms other than Windows 95/NT, `LoadExternalModuleEx` works exactly like `LoadExternalModule`.

Parameters

Input

Name	Type	Description
pathName	string	Relative or absolute pathname of the module to load.
callingModuleHandle	void pointer	Usually, the module handle of the calling DLL. You can use `__CVIUserHInst`. Zero indicates the project or executable.

Return Value

Name	Type	Description
moduleId	integer	ID of the loaded module.

Return Codes

Same as the return codes for `LoadExternalModule`.

Using this Function

Refer to the function help for `LoadExternalModule` for more information on this function.

If you call `LoadExternalModule` on an object or library module, `LoadExternalModule` resolves external references using symbols defined in the project or exported by object, library, or DLL import library modules that you have already loaded using `LoadExternalModuleEx` or `LoadExternalModule`. This is true even if you call `LoadExternalModule` in a DLL.

You might want to load an object or library module in a DLL and have the module link back to symbols that you defined in, but did not export from, the DLL. You can do this using `LoadExternalModuleEx`. You must specify the module handle of the DLL as **callingModuleHandle**. You can do so by using the LabWindows/CVI pre-defined variable `__CVIUserHInst`.

`LoadExternalModuleEx` first searches the global DLL symbols to resolve external references. `LoadExternalModuleEx` resolves any remaining unresolved references by searching the symbols defined in the project or exported by object, library, or import library modules that you have already loaded using `LoadExternalModule` or `LoadExternalModuleEx`.

`LoadExternalModuleEx` expects the DLL to contain a table of symbols to use to resolve references. If you create the DLL in LabWindows/CVI, the table is included automatically. If you create the DLL using an external compiler, you must arrange for the table to be included in the DLL. You can do this by creating an include file that includes all the symbols that the table must contain. You can then use the **External Compiler Support** command in the **Build** menu of the Project window to create an object file that contains the table. You must include the object file in the external compiler project you use to create the DLL.

`LoadExternalModuleEx` acts identically to `LoadExternalModule` in the following cases:

- You pass zero for **callingModuleHandle**.
- You pass `__CVIUserHInst` for **callingModuleHandle**, but you call the function from a file that is in the project or your executable, rather than in a DLL.
- You are not running under Windows 95/NT.

You cannot load the same external module using two different calling module handles. The function reports an error if you attempt to load an external module when you have already loaded it under a different module handle.

MakeDir

```
int result = MakeDir (char directoryName[]);
```

Purpose

Creates a new directory with the name you specify.

Note ***You can create only one directory at a time.***

Parameter

Input

Name	Type	Description
directoryName	string	New directory name.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	One of the path components not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete the operation.
`-5`	Invalid path; for example, `c:filename` in Windows.
`-6`	Access denied.
`-8`	Disk is full.
`-9`	Directory or file already exists with same pathname.

Example

```
/* Make a new directory named \DATA\WAVEFORM on drive C */
/* assuming that C:\DATA does not exist */
MakeDir ("C:\\DATA");
MakeDir ("C:\\DATA\\WAVEFORM");
```

MakePathname

```
void MakePathname (char directoryName[], char fileName[], char pathName[]);
```

Purpose

Constructs a pathname from a directory path and a filename. The subroutine ensures that a backslash separates the directory path and the filename.

Parameters

Input

Name	Type	Description
directoryName	string	Directory path.
fileName	string	Base filename and extension.

Output

Name	Type	Description
pathName	string	Pathname

Return Value

None.

Parameter Discussion

pathName must be at least `MAX_PATHNAME_LEN` bytes long. If `MakePathname` constructs a pathname that exceeds that size, it fills in **pathName** with an empty string instead.

Example

```
char dirname[MAX_PATHNAME_LEN];
char pathname[MAX_PATHNAME_LEN];
GetProjectDir (dirname);
MakePathname (dirname, "FILE.DAT", pathname);
```

MapPhysicalMemory

```
int status = MapPhysicalMemory (unsigned int physAddress,
                                unsigned int numBytes, void *ptrToMappedAddr,
                                int *mapHandle);
```

Note ***Only the Windows 95/NT versions of LabWindows/CVI support*** `MapPhysicalMemory`**.**

Purpose

Maps a physical address to a pointer that you can use in your program like any other C pointer. For example, you can read or write an area of physical memory by incrementing the pointer after each access.

In cases where you cannot transfer all your data at once, `MapPhysicalMemory` provides better performance than `ReadPhysicalMemory` or `WritePhysicalMemory`. There is a significant performance penalty to mapping and unmapping physical memory. If you call `ReadPhysicalMemory` or `WritePhysicalMemory` on each access, you are mapping and unmapping the memory each time.

When you no longer need the pointer, call `UnMapPhysicalMemory` on the handle **mapHandle** returns.

Note ***Under Windows 95/NT,*** `MapPhysicalMemory` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`**.**

Parameters

Input

Name	Type	Description
physAddress	unsigned integer	Physical address to map into user memory.
numBytes	unsigned integer	Number of bytes of physical memory to map.

Output

Name	Type	Description
ptrToMappedAddr	any type of pointer	Contains the mapped physical address. Pass a pointer by reference as this parameter.
mapHandle	integer	Contains the handle that you pass to the `UnMapPhysicalMemory` function to unmap the physical memory.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
`1`	Success.
`0`	**numBytes** is `0`, memory allocation failed, the operating system reported an error, or the low-level support driver is not loaded.

Parameter Discussion

No restrictions exist on the value of **physAddress**. It can be below or above 1 MB.

Note `MapPhysicalMemory` ***does not check the validity of the physical address.***

Example

```
int physAddr = 0xB000;
int numBytes = 0x1000;
int *physMemPtr;
int mapHandle;
int data, i;

if  ( ! MapPhysicalMemory (physAddr, numBytes, &physMemPtr,
                                    &mapHandle))
   {
   /* report error */
   }
```

```
else
    {
    for (i=0; i < numBytes/sizeof(int); i++)
        {
        /* <determine data to write> */
        *physMemPtr++ = data;
        }
    UnMapPhysicalMemory (mapHandle);
    }
```

See Also

UnMapPhysicalMemory, CVILowLevelSupportDriverLoaded

outp

```
char byteWritten = outp(int portNumber, char byteToWrite);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `outp`.

Purpose

Writes a byte to a port.

Note ***Under Windows NT,*** `outp` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`.

Parameters

Input

Name	Type	Description
portNumber	integer	Port.
byteToWrite	char	Byte to write.

Return Value

Name	Type	Description
byteWritten	char	Byte that you wrote.

See Also

`CVILowLevelSupportDriverLoaded`

outpw

```
short wordWritten = outpw (short portNumber, int wordToWrite);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `outpw`***.***

Purpose

Writes a word to a port.

Note ***Under Windows NT,*** `outpw` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`***.***

Parameters

Input

Name	Type	Description
portNumber	integer	Port.
wordToWrite	short	Word to write.

Return Value

Name	Type	Description
wordWritten	short	Word that you wrote.

See Also

`CVILowLevelSupportDriverLoaded`

ReadFromPhysicalMemory

```
int status = ReadFromPhysicalMemory (unsigned int physicalAddress,
                                     void *destinationBuffer,
                                     unsigned int numberOfBytes);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `ReadFromPhysicalMemory`**.**

Purpose

Copies the contents of a region of physical memory into **destinationBuffer**. `ReadFromPhysicalMemory` does not check whether the memory actually exists. If the memory does not exist, `ReadFromPhysicalMemory` returns the success value but does not read data.

Note ***Under Windows 95/NT,*** `ReadFromPhysicalMemory` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`**.**

Parameters

Input

Name	Type	Description
physicalAddress	unsigned integer	Physical address to read from. No restrictions exists on the address; it can be below or above 1 MB.
destinationBuffer	void pointer	Buffer into which to copy the physical memory.
numberOfBytes	unsigned integer	Number of bytes to copy from physical memory.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
1	Success.
0	Operating system reported failure, or low-level support driver not loaded.

See Also

ReadFromPhysicalMemoryEx, MapPhysicalMemory, CVILowLevelSupportDriverLoaded

ReadFromPhysicalMemoryEx

```
int status = ReadFromPhysicalMemoryEx (unsigned int physicalAddress,
                                       void *destinationBuffer,
                                       unsigned int numberOfBytes, int bytesAtATime);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `ReadFromPhysicalMemoryEx`**.**

Purpose

Copies the contents of a region of physical memory into the buffer you specify. It can copy the data in units of 1, 2, or 4 bytes at a time. `ReadFromPhysicalMemoryEx` does not check whether the memory actually exists. If the memory does not exist, `ReadFromPhysicalMemoryEx` returns the success value but does not read data.

Note ***Under Windows 95/NT,*** `ReadFromPhysicalMemoryEx` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`**.**

Parameters

Input

Name	Type	Description
physicalAddress	unsigned integer	Physical address to read from. No restrictions exist on the address; it can be above or below 1 MB.
destinationBuffer	void pointer	Buffer into which to copy the physical memory.
numberOfBytes	unsigned integer	Number of bytes to copy from physical memory.
bytesAtATime	integer	Unit size in which to copy the data; Can be 1, 2, or 4 bytes.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
1	Success.
0	Operating system reported failure, low-level support driver not loaded, **numberOfBytes** is not a multiple of **bytesAtATime**, or invalid value for **bytesAtATime**.

Parameter Discussion

numberOfBytes must be a multiple of **bytesAtATime**.

See Also

`MapPhysicalMemory`, `CVILowLevelSupportDriverLoaded`

ReleaseExternalModule

```
int status = ReleaseExternalModule (int moduleID);
```

Purpose

Decreases the reference count for a module you load using `LoadExternalModule` or `LoadExternalModuleEx`.

When you call `LoadExternalModule` successfully on a module, the module reference count increments by one. When you call `ReleaseExternalModule`, the module reference count decrements by one.

If the reference count decreases to zero, the module ID is invalidated and you cannot access the module through `GetExternalModuleAddr` or `RunExternalModule`. If, in addition, the module file is not in the project and not loaded as an instrument, `ReleaseExternalModule` removes the external module from memory.

If you want to unload the module regardless of the reference count, call `UnloadExternalModule` rather than `ReleaseExternalModule`. Use `ReleaseExternalModule` when multiple calls might have been made to `LoadExternalModule` on the same module and you do not want to unload the module in case other parts of the application still use it.

Parameter

Input

Name	Type	Description
moduleID	integer	Module ID you obtain from `LoadExternalModule` or `LoadExternalModuleEx`.

Return Value

Name	Type	Description
status	integer	Indicates the result of the operation.

Return Codes

Code	Description
`> 0`	Success, but the module was not unloaded. The value indicates the number of remaining references.
`0`	Success, and the module was unloaded.
`-5`	Module cannot be unloaded because another external module that is currently loaded references it.
`-9`	Invalid module ID.

RenameFile

```
int result = RenameFile (char existingFileName[], char newFileName[]);
```

Purpose

Renames an existing file.

Parameters

Input

Name	Type	Description
existingFileName	string	Existing filename.
newFileName	string	New filename.

Return Value

Name	Type	Description
result	integer	Result of rename operation.

Return Codes

Code	Description
0	Success.
-1	File not found or directory in path not found.
-3	General I/O error occurred.
-4	Insufficient memory to complete the operation.
-5	Invalid path, for either of the filenames.
-6	Access denied.
-7	Specified existing path is a directory, not a file.
-8	Disk is full.
-9	New file already exists.

Parameter Discussion

existingFileName and **newFileName** can contain DOS wildcard characters '?' and '*'. If **existingFileName** has wildcards, `RenameFile` renames all matching files. If **newFileName** has wildcards, `RenameFile` matches it to **existingFileName**.

existingFileName can be the empty string `""`, in which case `RenameFile` renames the file that the most recent call to `GetFirstFile` or `GetNextFile` found.

Under Windows, if the arguments to `RenameFile` specify files on different disk drives, `RenameFile` copies the source to the target and then deletes the source file.

Under UNIX, if the arguments to `RenameFile` specify files on different file systems, `RenameFile` copies the source to the target and then deletes the source file.

RetireExecutableHandle

```
int status = RetireExecutableHandle (int executableHandle);
```

Purpose

Informs the LabWindows/CVI Utility Library that you no longer intend to use the handle you acquired from `LaunchExecutableEx`. When you call `RetireExecutableHandle`, the LabWindows/CVI Utility Library can reuse the memory allocated to keep track of the state of the executable.

When you launch several processes with `LaunchExecutableEx` but do not call `RetireExecutableHandle` on them, you might reach the limit for the maximum number of processes the system imposes. This happens even if the processes terminate; the program does not recognize that the processes have terminated until you call `RetireExecutableHandle`.

Parameter

Input

Name	Type	Description
executableHandle	integer	Executable handle you obtain from `LaunchExecutableEx`. `-1` = handle is invalid `0` = success

Return Value

Name	Type	Description
status	integer	Result of the operation.

RoundRealToNearestInteger

```
long n = RoundRealToNearestInteger (double inputRealNumber);
```

Purpose

Rounds its floating-point argument and returns the result as a long integer. A value with a fractional part of exactly 0.5 is rounded to the nearest even number.

Parameter

Input

Name	Type	Description
inputRealNumber	double-precision	Real number to round.

Return Value

Name	Type	Description
n	long	Result of the rounding operation.

Example

```
long n;
n = round (1.2);  /* result: 1L  */
n = round (1.8);  /* result: 2L  */
n = round (1.5);  /* result: 2L  */
n = round (0.5);  /* result: 0L  */
n = round (-1.2); /* result: -1L */
n = round (-1.8); /* result: -2L */
n = round (-1.5); /* result: -2L */
n = round (-0.5); /* result: 0L  */
```

RunExternalModule

```
int result = RunExternalModule (int moduleID, char *buffer);
```

Purpose

Calls the pre-defined entry point function in an external module. Refer to `LoadExternalModule`.

Parameters

Input

Name	Type	Description
moduleID	integer	ID of loaded module.
buffer	string	Parameter buffer.

Return Value

Name	Type	Description
result	integer	Indicates the result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	Out of memory.
`-3`	Entry point is undefined.
`-4`	Invalid file format.
`-5`	Undefined references.
`-8`	Cannot open file.
`-9`	Invalid module ID.

Parameter Discussion

moduleID is the value `LoadExternalModule` returns. **buffer** is a character array in which you can pass information to and from the module.

`RunExternalModule` requires that the module define the following function:

```
void _xxx_entry_point (char [])
```

where `xxx` is the base name of the file, in lowercase. For example, if the pathname of the file is

```
C:\LW\PROGRAMS\TEST01.OBJ,
```

the name of the entry point must be

```
_test01_entry_point.
```

Example

```
int module_id;
int status;
char *pathname;
pathname = "EXTMOD.OBJ";
module_id = LoadExternalModule (pathname);
if (module_id <0)
   FmtOut ("Unable to load %s\n", pathname);
else {
   RunExternalModule (module_id, "");
   UnloadExternalModule (module_id);
}
```

SetBreakOnLibraryErrors

```
int oldState = SetBreakOnLibraryErrors (int newState);
```

Purpose

When you enable debugging and a National Instruments library function reports an error, LabWindows/CVI can display a runtime error dialog box and suspend execution. You can use `SetBreakOnLibraryErrors` to enable or disable this feature.

In general, it is best to use the Break on Library Errors checkbox in the **Run Options** command of the Project window to enable or disable this feature. You should use this function only when you want to temporarily disable the Break on Library Errors feature around a segment of code.

`SetBreakOnLibraryErrors` does not affect the state of the Break on Library Errors checkbox in the **Run Options** command of the Project window.

If you disable debugging, `SetBreakOnLibraryErrors` has no effect. LabWindows/CVI never reports run-time errors when you disable debugging.

Parameter

Input

Name	Type	Description
newState	integer	Pass a nonzero value to enable. Pass zero to disable.

Return Value

Name	Type	Description
oldState	integer	Previous state of the Break on Library Errors feature.

Return Codes

Code	Description
`1`	Previously enabled.
`0`	Previously disabled, or debugging is disabled.

Example

```
int oldValue;
oldValue = SetBreakOnLibraryErrors (0);
/* Function calls that may legitimately return errors. */
SetBreakOnLibraryErrors (oldValue);
```

See Also

GetBreakOnLibraryErrors

SetBreakOnProtectionErrors

```
int oldState = SetBreakOnProtectionErrors (int newState);
```

Purpose

If you enable debugging, LabWindows/CVI uses information it gathers from compiling your source code to make extensive run-time checks to protect your program. When it encounters a protection error at run-time, LabWindows/CVI displays a dialog box and suspends execution.

Examples of protection errors include:

- Dereferencing an invalid pointer value in source code.
- Attempting, in source code, to read or write beyond the end of an array.
- Making a function call, in source code, in which an array is smaller than the function expects.
- Performing pointer arithmetic in source code which generates an invalid address.

You can use `SetBreakOnProtectionErrors` to prevent LabWindows/CVI from displaying the dialog box and suspending execution when it encounters a protection error. In general, it is better not to disable the Break on Protection Errors feature. Nevertheless, you might want to disable it temporarily around a line of code for which LabWindows/CVI erroneously reports a protection error.

If you disable debugging, `SetBreakOnProtectionErrors` has no effect. LabWindows/CVI never reports run-time errors when you disable debugging.

☞ **Note** ***If an invalid memory access generates a processor exception, LabWindows/CVI reports the error and terminates your program regardless of the debugging level or the state of the Break on Protection Errors feature.***

Parameter

Input

Name	Type	Description
newState	integer	Pass a nonzero value to enable. Pass zero to disable.

Return Value

Name	Type	Description
oldState	integer	Previous state of the Break on Protection Errors feature.

Return Codes

Code	Description
1	Previously enabled.
0	Previously disabled, or debugging is disabled.

Example

```
int oldValue;
oldValue = SetBreakOnProtectionErrors (0);
/* Statement that erroneously reports an error */
SetBreakOnProtectionErrors (oldValue);
```

See Also

GetBreakOnProtectionErrors

SetDir

```
int result = SetDir (char directoryName[]);
```

Purpose

Sets the current working directory to the directory you specify. Under Windows 3.1, `SetDir` can change the current working directory on any drive; however, it does not change the default drive. To change the default drive, use `SetDrive`.

Parameter

Input

Name	Type	Description
directoryName	string	New current working directory.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	Specified directory not found or out of memory.

Parameter Discussion

Under Windows 3.1, **directoryName** must not contain a drive letter.

SetDrive

```
int result = SetDrive (int driveNumber);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `SetDrive`**.**

Purpose

Sets the current default drive.

Parameter

Input

Name	Type	Description
driveNumber	integer	New drive number; 0–25.

Return Value

Name	Type	Description
result	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	Invalid drive number.

Using This Function

The mapping between the drive number and the logical drive letter is 0 = A, 1 = B, and so on.

SetFileAttrs

```
int result = SetFileAttrs (char fileName[], int read-only, int system,
                           int hidden, int archive);
```

☞ **Note** ***Only the Windows versions of LabWindows/CVI support*** `SetFileAttrs`**.**

Purpose

Sets the **read-only**, **system**, **hidden** and **archive** attributes of a file.

The **read-only** attribute protects a file from being overwritten and prevents the creation of a file with the same name.

The **system** attribute and **hidden** attribute both prevent the file from appearing in a directory list and exclude it from normal searches.

The operating system sets the **archive** attribute whenever you modify the file. The DOS `backup` command clears it.

Parameters

Input

Name	Type	Description
fileName	string	File to set the attributes of.
read-only	integer	Read-only attribute.
system	integer	System attribute.
hidden	integer	Hidden attribute.
archive	integer	Archive attribute.

Return Value

Name	Type	Description
result	return value	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File not found, or attribute cannot be changed.

Parameter Discussion

Each attribute parameter can have one of the following values:

`0` = clears the attribute

`1` = sets the attribute

`-1` = leaves the attribute unchanged

fileName can be the empty string `""`, in which case `SetFileAttrs` sets the attributes of the file that the most recent call to `GetFirstFile` or `GetNextFile` found.

SetFileDate

```
int status = SetFileDate (char fileName[], int month, int day, int year);
```

Purpose

Sets the date of a file.

Parameters

Input

Name	Type	Description
fileName	string	File to set the attributes of.
month	integer	Month; 1–12
day	integer	Day of month, 1–31
year	integer	Year; 1980–2099

Return Value

Name	Type	Description
status	integer	Result of the operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File not found or directory in path not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete the operation.
`-5`	Invalid date or invalid path; for example, `c:filename` in Windows.
`-6`	Access denied.

Parameter Discussion

fileName can be the empty string `""`, in which case `SetFileAttrs` sets the date of the file that the most recent call to `GetFirstFile` or `GetNextFile` found.

SetFileTime

```
int result = SetFileTime (char fileName[], int hours, int minutes,
                          int seconds);
```

Purpose

Sets the time of a file.

Parameters

Input

Name	Type	Description
fileName	string	File to set the time of.
hours	integer	Hours; 0–23
minutes	integer	Minutes; 0–59
seconds	integer	Seconds; 0–58, odd values are rounded down.

Return Value

Name	Type	Description
result	integer	Result of operation.

Return Codes

Code	Description
`0`	Success.
`-1`	File not found or directory in path not found.
`-3`	General I/O error occurred.
`-4`	Insufficient memory to complete the operation.
`-5`	Invalid time, or invalid path; for example, `c:filename` in Windows.
`-6`	Access denied.

Parameter Discussion

fileName can be the empty string `""`, in which case `SetFileTime` sets the time of the file that the most recent call to `GetFirstFile` or `GetNextFile` found.

If you enter an odd number for the **seconds** parameter, the operating system rounds the value down to an even number.

SetPersistentVariable

```
void SetPersistentVariable (int value);
```

Purpose

Lets you store an integer value across multiple builds and executions of your project in the LabWindows/CVI development environment. When you unload a project or load a new project, `SetPersistentVariable` resets the value to zero.

`SetPersistentVariable` is useful when your program performs an action, such as setting up your instruments, that takes a long time and that you do not want to repeat each time you run your program. LabWindows/CVI initializes global variables in your program each time you run your project. Therefore, you cannot use them to indicate that you have already taken the action once.

To get around this problem, LabWindows/CVI maintains an integer variable across multiple builds and executions of your project. `SetPersistentVariable` sets the value of that variable. To retrieve the variable value, call `GetPersistentVariable()`.

Parameter

Input

Name	Type	Description
value	integer	Value to assign to the persistent variable.

Return Value

None.

SetStdioPort

```
int status = SetStdioPort (int stdioPort);
```

Note ***Only the UNIX versions of LabWindows/CVI support*** `SetStdioPort`**.**

Purpose

Sets the current destination for data written to the Standard Output and the source of data read from Standard Input.

You can specify either the LabWindows/CVI Standard Input/Output window or the Standard Input/Output of the host system.

Parameter

Input

Name	Type	Description
stdioPort	integer	`CVI_STDIO_WINDOW` (0) = LabWindows/CVI Standard Input/Output window `HOST_SYSTEM_STDIO` (1) = host system's Standard Input/Output

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
`0`	Success.
`-2`	Destination was not a valid range.

Parameter Discussion

In a standalone executable, the default value for **stdioPort** is `CVI_STDIO_WINDOW`.

In the LabWindows/CVI development system, the default value for **stdioPort** is the current state of the Use Host System's Standard Input/Output option in the Environment dialog box of the Project window. The value you set using `SetStdioPort` is reflected the next time you open the Environment dialog box.

SetStdioWindowOptions

```
int status = SetStdioWindowOptions (int maxNumLines,
                                    int bringToFrontWhenModified,
                                    int showLineNumbers);
```

Purpose

Sets the current value of the following Standard Input/Output window options:

- Maximum number of lines
- Bring to front when modified
- Show line numbers

Parameters

Input

Name	Type	Description
maxNumLines	integer	Maximum number of lines you can store in the Standard Input/Output window. If this amount is exceeded, lines are discarded from the top. Valid range = 100 to 1,000,000
bringToFrontWhenModified	integer	Indicates whether to bring the Standard Input/Output window to the front each time you add a string or character to it. `1` = Yes `0` = No
showLineNumbers	integer	Indicates whether to show line numbers in the Standard Input/Output window. `1` = Yes `0` = No

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
`0`	Success.
`-1`	Maximum number of lines is not within the valid range.

Parameter Discussion

In an executable, the default value of **maxNumLines** is 10,000. In the LabWindows/CVI development system, the default value is the value of the Maximum Number of Lines in Standard Input/Output Window control in the Environment dialog box of the Project window. The value you set using `SetStdioWindowOptions` is reflected the next time you bring up the Environment dialog box.

In an executable, the default value of **bringToFrontWhenModified** is `1`. In the LabWindows/CVI development system, the default value is the state of the Bring Standard Input/Output Window to Front When Modified checkbox in the Environment dialog box of the Project window. The value you set using `SetStdioWindowOptions` is reflected the next time you bring up the Environment dialog box.

In an executable, the default value of **showLineNumbers** is 0. In the LabWindows/CVI development system, the default value is the current state you set by selecting **View»Line Numbers** in the Standard Input/Output Window. The value you set using `SetStdioWindowOptions` is reflected the next time you bring up the **View** menu.

SetStdioWindowPosition

```
int status = SetStdioWindowPosition (int top, int left);
```

Purpose

Sets the current position, in pixels, of the client area of the Standard Input/Output window relative to the upper left corner of the screen. The client area begins under the title bar and to the right of the frame.

Parameters

Input

Name	Type	Description
top	integer	Distance, in pixels, of the top of client area of the Standard Input/Output window relative to the top of the screen. Valid range = `VAL_AUTO_CENTER`; -16,000 to +16,000
left	integer	Distance, in pixels, of the left edge of client area of the Standard Input/Output window relative to the left edge of the screen. Valid range = `VAL_AUTO_CENTER`; -16,000 to +16,000

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
`0`	Success.
`-1`	**top** is not within the valid range.
`-2`	**left** is not within the valid range.

Parameter Discussion

To vertically center the Standard Input/Output window client area within the area of the screen, pass `VAL_AUTO_CENTER` as the **top** parameter.

To horizontally center the Standard Input/Output window client area within the area of the screen, pass `VAL_AUTO_CENTER` as the **left** parameter.

SetStdioWindowSize

```
int status = SetStdioWindowSize (int height, int width);
```

Purpose

Sets the height and width, in pixels, of the client area of the Standard Input/Output window. The client area excludes the frame and the title bar.

Parameters

Input

Name	Type	Description
height	integer	Height, in pixels, of the client area of the Standard Input/Output window. Valid range = 0 to 16,000
width	integer	Width, in pixels, of the client area of the Standard Input/Output window. Valid range = 0 to 16,000

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
`0`	Success.
`-1`	**height** is not within the valid range.
`-2`	**width** is not within the valid range.
`-3`	You must call this function from the main thread of your program.

SetStdioWindowVisibility

```
void SetStdioWindowVisibility (int visible);
```

Purpose

Brings to the front or hides the Standard Input/Output window.

Parameter

Input

Name	Type	Description
visible	integer	`1` = bring to front `0` = hide

Return Value

None.

SetSystemDate

```
int status = SetSystemDate (int month, int day, int year);
```

☞ **Note** ***Only the Windows versions of LabWindows/CVI support*** `SetSystemDate`***. Under Windows NT, you must have system administrator status to use this function.***

Purpose

Sets the system date.

Parameters

Input

Name	Type	Description
month	integer	Month; 1–12.
day	integer	Day of month; 1–31.
year	integer	Year; under Windows 3.1, the year is limited to the values 1980–2099.

Return Value

Name	Type	Description
status	integer	Success or failure.

Return Codes

Code	Description
`0`	Success.
`-1`	Operating system reported failure, probably because of an invalid parameter.

SetSystemTime

```
int status = SetSystemTime(int hours, int minutes, int seconds);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `SetSystemTime`***. Under Windows NT, you must have system administrator status to use this function.***

Purpose

Sets the system time.

Parameters

Input

Name	Type	Description
hours	integer	Hours; 0–23.
minutes	integer	Minutes; 0–59.
seconds	integer	Seconds; 0–58. Odd values are rounded down.

Return Value

Name	Type	Description
status	integer	Success or failure.

Return Codes

Code	Description
`0`	Success.
`-1`	Operating system reported failure, probably because of an invalid parameter.

SplitPath

```
void SplitPath (char pathName[], char driveName[], char directoryName[],
                          char fileName[]);
```

Purpose

Splits a pathname into the drive name, the directory name, and the filename.

Parameters

Input

Name	Type	Description
pathName	string	Pathname to split.

Output

Name	Type	Description
driveName	string	Drive name.
directoryName	string	Full directory path, ending with directory separator character.
fileName	string	Simple filename.

Return Value

None.

Parameter Discussion

driveName, **directoryName**, and **fileName** can each be `NULL`. If not `NULL`, they must be buffers of the following size or greater:

driveName `MAX_DRIVENAME_LEN`

directoryName `MAX_DIRNAME_LEN`

fileName `MAX_FILENAME_LEN`

On operating systems without drive names (such as UNIX), `SplitPath` always fills **driveName** with the empty string.

Example

```
char pathName[MAX_PATHNAME_LEN];
char driveName[MAX_DRIVENAME_LEN];
char dirName[MAX_DIRNAME_LEN];
char fileName[MAX_FILENAME_LEN];
```

```
SplitPath (pathName, driveName, dirName, fileName);
/* If pathName contains
       c:\cvi\samples\apps\update.c
   then
       driveName contains         "c:"
       dirName contains           "\cvi\samples\apps\"
       fileName contains          "update.c"

   If pathName is
       \\computer\share\dirname\foo.c
   then
       drive name is              ""
       directory name is          "\\computer\share\dirname\"
       filename is                "foo.c" */
```

SyncWait

```
void SyncWait (double beginTime, double interval);
```

Purpose

Waits until **interval** seconds elapse since **beginTime**.

Parameters

Input

Name	Type	Description
beginTime	double-precision	Value `Timer` returns.
interval	double-precision	Number of seconds to wait after **beginTime**.

Return Value

None.

Parameter Discussion

beginTime must be a value `Timer` returns.

The resolution on Windows is normally 1 ms. If, however, you set the `useDefaultTimer` configuration option to `True`, the resolution is 55 ms.

The resolution on Sun Solaris is 1 ms.

SystemHelp

```
int status = SystemHelp (char helpFile[], unsigned int command,
                         unsigned long additionalLongData,
                         char additionalStringData[]);
```

Purpose

Under Windows, starts Windows Help, `winhelp.exe`, on a help file you specify. You can pass optional data that indicates the nature of the help you want to display.

For information about creating help files, refer to the Microsoft Windows programming documentation, which is not included with LabWindows/CVI.

Under UNIX, `SystemHelp` starts the HyperHelp help viewer on the help file you specify.

Note ***Although you may use the HyperHelp viewer that comes with LabWindows/CVI, you may not distribute it in an application you build unless you purchase a license from Bristol Technology, Inc.***

Parameters

Input

Name	Type	Description
helpFile	string	Points to a string that contains the name of the help file you want to display.
command	unsigned integer	Specifies the type of help you want to display.
additionalLongData	unsigned long integer	Depends on the **command** parameter as described in the following *Parameter Discussion*.
additionalStringData	string	Depends on the **command** parameter as described in the following *Parameter Discussion*.

Return Value

Name	Type	Description
status	integer	Nonzero on success; zero on failure.

Parameter Discussion

helpFile contains the pathname of the file you want to display. The pathname can be followed by an angle bracket (>) and the name of a secondary window if you want the topic to appear in a secondary window rather than in the primary window. The [`WINDOWS`] section of the help project (`.hpj`) file must define the name of the secondary window.

command can be one of the following values:

`HELP_COMMAND`—Executes a help macro or macro string. In this case, **additionalStringData** is the help macro to execute.

`HELP_CONTENTS`—Displays the help contents topic as defined by the contents option in the [`OPTIONS`] section of the `.hpj` file. `HELP_CONTENTS` is for backward compatibility. New programs should provide a `.cnt` file and use the `HELP_FINDER` command.

`HELP_CONTEXT`—Displays help for a particular topic identified by a context number that has been defined in the [`MAP`] section of the `.hpj` file. In this case, **additionalLongData** is the context number of the topic.

`HELP_CONTEXTPOPUP`—Displays in a pop-up window a particular help topic identified by a context number that has been defined in the [`MAP`] section of the `.hpj` file. `HELP_CONTEXTPOPUP` does not display the main help window. In this case, **additionalLongData** is the context number of the topic.

`HELP_FINDER`—Displays the Help Topics dialog box.

`HELP_HELPONHELP`—Displays the contents topic of the Using Help file if it is available.

`HELP_KEY`—Displays the topic in the keyword list that matches the keyword passed in the **additionalStringData** parameter if one exact match exists. Under Windows 3.1, if more than one match exists, `HELP_KEY` displays the first topic found. If no match exists `HELP_KEY` displays an error message. Under Windows 95/NT, if more than one match exists, `HELP_KEY` displays the Topics Found dialog box.

`HELP_PARTIALKEY`—Displays the topic found in the keyword list that matches the keyword passed in the **additionalStringData** parameter if one exact match exists.

Under Windows 3.1, if more than one match exists, `HELP_PARTIALKEY` displays the Search dialog box with the topics listed in the Go To list box. If no match exists, `HELP_PARTIALKEY` displays the Search dialog box. If you want to bring up just the Search dialog box without passing a keyword, pass an empty string (`""`).

Under Windows 95/NT, if more than one match exists, `HELP_PARTIALKEY` displays the Topics Found dialog box. If you want to display the Index without passing a keyword, pass an empty string (`""`).

`HELP_POPUPID`—Displays in a pop-up window the topic identified by a context string. `HELP_POPUPID` does not display the main help window.

`HELP_QUIT`—Closes the help file. `HELP_QUIT` has no effect if another executable opens the help file.

`HELP_SETCONTENTS`—Determines which Contents topic help appears when the user chooses the Contents button in a help window if the help file does not have an associated `.cnt` file. If a help file has two or more Contents topics, you must assign one as the default. Call `SystemHelp` with **command** set to `HELP_SETCONTENTS` and the **additionalLongData** parameter specifying the corresponding context identifier.

TerminateExecutable

```
int status = TerminateExecutable (int executableHandle);
```

Purpose

Terminates an executable if it has not already terminated.

Windows 3.1 terminates an executable by sending close messages to each window in the application. If the application does not honor the close messages, the application does not terminate. `TerminateExecutable` gives up control for a limited time to give the application an opportunity to process the close messages. This time period should be sufficient for all applications. When you must allow more time, your program can call `ProcessSystemEvents` in a loop, as shown in the following example.

Example for Windows 3.1

```
#define TIME_LIMIT 5.0            /* number of seconds */
double startTime;
startTime = Timer ();
TerminateExecutable (handle);
while (!ExecutableHasTerminated(handle)
      && (Timer()-startTime > TIME_LIMIT))
   ProcessSystemEvents();
```

Parameter

Input

Name	Type	Description
executableHandle	integer	Executable handle acquired from `LaunchExecutableEx`.

Return Value

Name	Type	Description
status	integer	Result of the operation.

Return Codes

Code	Description
`-1`	Handle is invalid.
`0`	Handle is valid. To determine if the function actually terminated the executable, use `ExecutableHasTerminated`.

Timer

```
double t = Timer (void);
```

Purpose

Returns the number of seconds that have elapsed since the first call to `Timer`, `Delay`, or `SyncWait` or the first operation on a timer control. The value is never reset to zero except when you restart your program. The resolution on Windows is normally 1 ms. If, however, you set the `useDefaultTimer` configuration option to `True`, the resolution is 55 ms.

The resolution on Sun Solaris is 1 ms.

Parameters

None.

Return Value

Name	Type	Description
t	double-precision	Number of seconds since the first call to `Timer`.

TimeStr

```
char *s = TimeStr (void);
```

Purpose

Returns an 8-character string in the form *HH:MM:SS*, where *HH* is the hour, *MM* is in minutes, and *SS* is in seconds.

Parameters

None.

Return Value

Name	Type	Description
s	8-character string	Time in *HH:MM:SS* format.

TruncateRealNumber

```
double y = TruncateRealNumber (double inputRealNumber);
```

Purpose

Truncates the fractional part of **inputRealNumber** and returns the result as a real number.

Parameter

Input

Name	Type	Description
inputRealNumber	double-precision	Real number to truncate.

Return Value

Name	Type	Description
y	double-precision	Value of **inputRealNumber** without its fractional part.

UnloadExternalModule

```
int status_id = UnloadExternalModule (int moduleID);
```

Purpose

Unloads an external module file you loaded using `LoadExternalModule`.

Parameter

Output

Name	Type	Description
moduleID	integer	ID of loaded module.

Return Value

Name	Type	Description
status_id	integer	Indicates the result of the operation.

Return Codes

Code	Description
`0`	Success.
`-9`	Failure because of an invalid `module_id`.

Parameter Discussion

moduleID is the value `LoadExternalModule` returns, or `-1`. If **moduleID** is `-1`, LabWindows/CVI unloads all external modules.

Example

```
int module_id;
int status;
char *pathname;
pathname = "PROG.OBJ";
module_id = LoadExternalModule (pathname);
if (module_id <0)
   FmtOut ("Unable to load %s\n", pathname);
else {
   RunExternalModule (module_id, "");
   UnloadExternalModule (module_id);
}
```

UnMapPhysicalMemory

```
int status = UnMapPhysicalMemory (int mapHandle);
```

Note ***Only the Windows 95/NT versions of LabWindows/CVI support*** `UnMapPhysicalMemory`**.**

Purpose

Unmaps an address that you mapped using `MapPhysicalMemory`. You do not pass the address to this function. Instead, you pass the handle that `MapPhysicalMemory` returned.

Note ***Under Windows 95/NT,*** `UnMapPhysicalMemory` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`**.**

Parameters

Input

Name	Type	Description
mapHandle	integer	Handle `MapPhysicalMemory` returns.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
1	Success.
0	`mapHandle` is not valid, the operating system reported an error, or the low-level support driver is not loaded.

See Also

`CVILowLevelSupportDriverLoaded`

WriteToPhysicalMemory

```
int status = WriteToPhysicalMemory (unsigned int physicalAddress,
                                    void *sourceBuffer, unsigned int numberOfBytes);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `WriteToPhysicalMemory`**.**

Purpose

Copies the contents of **destinationBuffer** into a region of physical memory. `WriteToPhysicalMemory` does not check whether the memory actually exists. If the memory does not exist, `WriteToPhysicalMemory` returns the success value but does not read any data.

Note ***Under Windows 95/NT,*** `WriteToPhysicalMemory` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`**.**

Parameters

Input

Name	Type	Description
physicalAddress	unsigned integer	Physical address to write to. No restrictions exist on the address; it can be below or above 1 MB.
sourceBuffer	void pointer	Buffer from which to copy the physical memory.
numberOfBytes	unsigned integer	Number of bytes to copy to physical memory.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
1	Success.
0	Operating system reported failure, or low-level support driver not loaded.

See Also

WriteToPhysicalMemoryEx, MapPhysicalMemory, CVILowLevelSupportDriverLoaded

WriteToPhysicalMemoryEx

```
int status = WriteToPhysicalMemoryEx (unsigned int physicalAddress,
                          void *sourceBuffer, unsigned int numberOfBytes,
                          int bytesAtATime);
```

Note ***Only the Windows versions of LabWindows/CVI support*** `WriteToPhysicalMemoryEx`**.**

Purpose

Copies the contents of the buffer you specify to a region of physical memory. It can copy the data in units of 1, 2, or 4 bytes at a time. `WriteToPhysicalMemoryEx` does not check whether the memory actually exists. If the memory does not exist, `WriteToPhysicalMemoryEx` returns the success value but does not read any data.

Note ***Under Windows 95/NT,*** `WriteToPhysicalMemoryEx` ***requires the LabWindows/CVI low-level support driver. LabWindows/CVI loads the driver at startup if it is on disk. You can check whether LabWindows/CVI loaded the driver at startup by calling*** `CVILowLevelSupportDriverLoaded`**.**

Parameters

Input

Name	Type	Description
physicalAddress	unsigned integer	Physical address to write to. No restrictions exist on the address; it can be above or below 1 MB.
sourceBuffer	void pointer	Buffer from which to copy the physical memory.
numberOfBytes	unsigned integer	Number of bytes to copy to physical memory.
bytesAtATime	integer	Unit size in which to copy the data; can be 1, 2, or 4 bytes.

Return Value

Name	Type	Description
status	integer	Indicates whether the function succeeded.

Return Codes

Code	Description
1	Success.
0	Operating system reported failure, low-level support driver not loaded, **numberOfBytes** is not a multiple of **bytesAtATime**, or invalid value for **bytesAtATime**.

Parameter Discussion

numberOfBytes must be a multiple of **bytesAtATime**.

See Also

`MapPhysicalMemory`, `CVILowLevelSupportDriverLoaded`

9

X Property Library

This chapter describes the functions in the Lab/Windows CVI X Property Library. The X Property Library contains functions that read and write properties to and from X Windows. The *X Property Library Overview* section contains general information about the X Property Library functions and panels. The *X Property Library Function Reference* section contains an alphabetical list of function descriptions.

These functions provide a mechanism for communication among X clients. This library provides capabilities similar to those available in the TCP Library, but differs from the TCP Library in the following significant ways:

- It conforms to a conventional method for X interclient communication.
- It works between any X clients that are connected to the same display and does not require any particular underlying communication protocol such as TCP.
- It provides a method for sharing data among X clients without explicit point-to-point connections between them.

X Property Library Overview

The X Property Library is available only in the UNIX versions of LabWindows/CVI. This section contains general information about the X Property Library functions and panels.

X Property Library Function Panels

The X Property Library function panels are grouped in the tree structure in Table 9-1 according to the types of operations they perform.

The first- and second-level headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels. Each X Property Library function panel generates an X Property Library function call.

Table 9-1. Functions in the X Property Library Function Tree

Class/Panel Name	Function Name
Accessing Remote Hosts	
Connect to X Server	`ConnectToXDisplay`
Disconnect from X Server	`DisconnectFromXDisplay`

Table 9-1. Functions in the X Property Library Function Tree (Continued)

Class/Panel Name	Function Name
Managing Property Types	
Create New Property Type	`CreateXPropType`
Get Property Type Name	`GetXPropTypeName`
Get Property Type Size	`GetXPropTypeSize`
Get Property Type Unit	`GetXPropTypeUnit`
Destroy Property Type	`DestroyXPropType`
Managing Property Information	
Create New Property	`CreateXProperty`
Get Property Name	`GetXPropertyName`
Get Property Type	`GetXPropertyType`
Destroy Property	`DestroyXProperty`
Accessing Window Properties	
Get Single Window Property Item	`GetXWindowPropertyItem`
Put Single Window Property Item	`PutXWindowPropertyItem`
Get Window Property Value	`GetXWindowPropertyValue`
Put Window Property Value	`PutXWindowPropertyValue`
Remove Window Property	`RemoveXWindowProperty`
Handling Property Events	
Install Property Callback	`InstallXPropertyCallback`
Uninstall Property Callback	`UninstallXPropertyCallback`
Get Error String	`GetXPropErrorString`

The online help with each panel contains specific information about operating each function panel.

X Interclient Communication

X applications often use X properties to communicate with each other. Properties are essentially tagged data associated with a window. Applications communicate by reading and writing properties to and from windows. In addition, an X application can request that the X server notify it whenever a specific property value changes on a window.

The X applications that need to communicate with each other must first connect to the same X display. Then they must agree upon the names and types of properties and the X window IDs that they use to transfer the data. Although it is a simple matter to agree upon the names and types of properties in advance, the X applications cannot know the window IDs in advance because they are different for each invocation of the program. There must be a mechanism for transferring the window IDs from one client to another. A client usually accomplishes this by placing a property that contains the window ID on the root window, which is a window that all clients can access. The window ID refers to the window that contains the data for transfer to other clients. The other clients read this property from the root window to determine where to access the data.

With the LabWindows/CVI X Property Library functions, you can connect to X displays and obtain the root window ID, read and write properties on windows, and monitor when specific properties change.

Property Handles and Types

Before you can read or write properties on windows, you must create the property and its type. `CreateXProperty` takes a property name and a property type and returns a property handle you can use to access properties on windows. The property type, which `CreateXPropType` creates, contains the attributes that determine how data for the property are stored and retrieved. More specifically, these attributes are the size and unit. The size is the number of bytes in a single property item. The unit is the number of bytes in the basic entities that make up a property item. Refer to `CreateXPropType` for more information on the meanings of the size and unit attributes.

Table 9-2 lists the three predefined property types that you do not have to create. These types are useful for defining properties to store X window IDs, integers, and strings.

Table 9-2. Predefined Property Types

Property Type	Name	Size/Unit
`WINDOW_X_PROP_TYPE`	`"WINDOW"`	`sizeof(WindowX)`
`INTEGER_X_PROP_TYPE`	`"INTEGER"`	`sizeof(int)`
`STRING_X_PROP_TYPE`	`"STRING"`	`sizeof(char)`

Communicating with Local Applications

You can use the function `ConnectToXDisplay` to connect to any X server on a network. However, if your program communicates only with other applications connected to the same display as LabWindows/CVI, you do not need to connect to the display using `ConnectToXDisplay`. Instead, use the global variable `CVIXDisplay`, which is a pointer to the X display that LabWindows/CVI uses. The variable `CVIXRootWindow` contains the X window ID of the root window of the display that LabWindows/CVI uses.

Hidden Window

Before you can read or write property data, you need the X window IDs of the windows that store the properties.

One option is to always use the root window ID for attaching properties. The variable `CVIXRootWindow` holds the root window ID for the local display. To get the root window ID for a remote display, call `ConnectToXDisplay`. This approach has disadvantages. First, if your program adds a property to the root window and does not delete it, the property remains

there indefinitely. Second, because only one root window exists, conflicts might arise when multiple applications attempt to access the same properties.

To overcome those disadvantages, LabWindows/CVI provides a hidden window. Before it runs your program, LabWindows/CVI creates a window that never displays. The variable `CVIXHiddenWindow` holds the X window ID for the hidden window. This window ID is always available to your program for reading and writing properties. When your program terminates, LabWindows/CVI removes the window and all its properties.

Property Callback Functions

You can use the X Property Library to instruct LabWindows/CVI to notify your program whenever there is a change to a property/set of properties on a window/set of windows. `InstallPropertyCallback` registers a function that LabWindows/CVI calls whenever any of the specified properties changes. The callback function must have the type `PropertyCallbackTypeX` as defined in `xproplib.h`. LabWindows/CVI passes the X display, window, and property that changed to the callback function. The **state** parameter of the callback function can be `NewValueX`, if the property value changed, or `DeleteX`, if the property was deleted. `UninstallPropertyCallback` disables the callback function.

Error Codes

`PropLibXErrType` is the data type of all return values in the X Property Library functions. `PropLibXErrType` is an enumerated (`enum`) type that contains descriptive constant names and numeric values for the errors. `PropLibXErrType` and its enumerated values are all integers. All error values are negative numbers.

The function descriptions in the following section include detailed descriptions of these error types.

Using the Library Outside of LabWindows/CVI

You can use the LabWindows/CVI X Property Library in applications developed outside of LabWindows/CVI. By linking your program with the library file `libxprop.a` in the `misc/lib` directory of the LabWindows/CVI installation directory, you can use all the X Property Library functions in your program. You cannot use the `libxprop.a` library within LabWindows/CVI. The following two functions are available only outside of LabWindows/CVI:

- `void _InitXPropertyLib(DisplayPtrX cviDisplay, WindowX rootWindow, WindowX hiddenWindow)`

  This function sets the global variables `CVIXDisplay`, `CVIXRootWindow`, and `CVIXHiddenWindow` of the X Property Library.

- `void HandlePropertyNotifyEvent(EventPtrX event)`

  This function calls the functions that you installed as property callbacks. You should call this function whenever you receive an `XPropertyNotify` event. The event must be a valid `XPropertyEvent`.

X Property Library Function Reference

This section describes each function in the LabWindows/CVI X Property Library in alphabetical order.

ConnectToXDisplay

```
PropLibXErrType status = ConnectToXDisplay (const char *displayName,
                           DisplayPtrX *display, WindowX *rootWindow);
```

Purpose

Connects to a remote X server.

Use `ConnectToXDisplay` to access an X server on a remote computer. This function returns a display pointer and the root window, which you can use to read and write properties on the root window of the remote X server.

If you want to communicate only with applications that use the same display as your application, you do not need this function. Instead, use the global variables `CVIXDisplay` and `CVIXRootWindow`, which contain the display and root window of the X server LabWindows/CVI uses.

Parameters

Input

Name	Type	Description
displayName	string	Determines the X server connection and which communication domain to use.

Output

Name	Type	Description
display	`DisplayPtrX`	Pointer to the display of the remote X server. Use this value as the argument to other library functions to communicate with the remote X server.
rootWindow	`WindowX`	Root window of the remote X server. Use this value as the parameter to other library functions to access properties on the root window of the remote X server.

Return Values

Constant Name	Value	Description
NoXErr	0	Function was successful.
InvalidParamXErr	-1	NULL was passed to one or more parameters.
TooManyConnectionsXErr	-6	Program has already made the maximum number of connections that the constant MAX_X_DISPLAYS defines. Use DisconnectFromXDisplay to allow more connections.
CannotConnectXErr	-7	Connection could not be made to the X server. This happens for a number of reasons, including an invalid display name, a network problem, or a security problem.

Parameter Discussion

Valid values for **displayName** include any valid arguments to the Xlib function XOpenDisplay. The format is hostname:server or hostname:server.screen, where:

- hostname specifies the name of the host computer to which the display is physically connected.
- server specifies the number of the server, usually 0, on its host computer.
- screen specifies the number of the default screen, usually 0, on the server.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about XOpenDisplay DefaultRootWindow.

CreateXProperty

```
PropLibXErrType status = CreateXProperty (const char *propertyName,
                           PropTypeHandleX propertyType,
                           PropertyHandleX *property);
```

Purpose

Creates X property information.

Use this function to define the attributes of the properties that you read and write on X windows. You must create properties with `CreateXProperty` before you can access them on X windows.

Each property has a unique name and a type, created by `CreateXPropType`, that you cannot change except by destroying the property and recreating it.

Note ***You can create a maximum of 256 different properties.***

Parameters

Input

Name	Type	Description
propertyName	string	Name of the property. Each property name is unique and has a type that you cannot change after you create the property.
propertyType	`PropTypeHandleX`	Type of the property. This value must be a predefined type or a value `CreateXPropType` returns.

Output

Name	Type	Description
property	`PropertyHandleX`	Handle to the property information created. Use this value as the parameter to other library functions to access the property on X windows.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to one or more parameters.
`InvalidPropTypeXErr`	`-5`	**propertyType** argument is not a valid property type. This value must be one of the predefined property types, or must be a value `CreateXPropType` returns.
`DupPropertyXErr`	`-8`	Property with the same **propertyName** but with a different **propertyType** already exists.
`InsuffMemXErr`	`-19`	Insufficient memory to store the property information, or 256 properties already exist.

Parameter Discussion

`CreateXProperty` sets the **propertyType** of the property the first time you write a property to a window. When you access a property on a window on which the property already exists, its type must match this value for the access to succeed.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about `XInternAtom`.

CreateXPropType

```
PropLibXErrType status = CreateXPropType (const char *typeName,
                           unsigned int size, unsigned int unit,
                           PropTypeHandleX *propertyType);
```

Purpose

Creates an X property type. You can use this function to define the attributes of the properties that you read and write on X windows. You must create property types with this function before you can create properties.

Each property type has a unique name and set of attributes that you cannot change except by destroying the property and recreating it.

Refer to Table 9-2 in the *Property Handles and Types* section of the *X Property Library Overview* section of this chapter for the three predefined property types, which you do not need to create using `CreateXPropType`.

Note ***You can create a maximum of 64 different property types.***

Parameters

Input

Name	Type	Description
typeName	string	Name of the property type. Each property type name is unique and has one set of attributes that you cannot change after you create the property type.
size	unsigned integer	Number of bytes in a single property item.
unit	unsigned integer	Number of bytes in the basic units that make up a property item.

Output

Name	Type	Description
propertyType	`PropTypeHandleX`	Property type created. Use this value as the type parameter to `CreateXProperty` to create properties.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to one or more parameters; **size** argument is 0; **unit** is not 1, 2, or 4; or **size** is not a multiple of **unit**.
`DupPropTypeXErr`	`-9`	Property type with the same **typeName** but with a different **size** or **unit** already exists.
`InsuffMemXErr`	`-19`	Insufficient memory to store the property information or 64 property types already exist.

Parameter Discussion

Usually, you can use the expression `sizeof(`*`type`*`)` for the **size** parameter, where *`type`* is the data type you use to store the property value. This value must be a multiple of the **unit** argument.

unit specifies how the X server views the property item, for example, as an array of 1-byte, 2-byte or 4-byte objects. **unit** is necessary to perform simple byte swapping between different types of computers. Refer to the following notes for more information.

If the property item consists of a single object, such as an integer or a character, the unit should be the size of the object. An exception is the `double` type, for which the default unit should be 4 bytes.

If the property item is a structure or array that contains a number of smaller objects, then the unit should be the number of bytes in the smaller objects.

☞ **Note** ***If you are communicating with a remote X server on a computer that uses different byte ordering than your application,*** `CreateXPropType` ***uses the unit you specify to perform the byte swapping. However, LabWindows/CVI cannot properly perform byte swapping for structures that contain different-sized members or for the*** `double` ***type. For these special cases, use a unit of one and explicitly perform byte swapping where you need to.***

☞ **Note** ***The LabWindows/CVI X Property Library specifies units in the number of*** *bytes* ***as opposed to*** *bits*. ***Thus, the format values of 8, 16, and 32 that Xlib functions use correspond to units of 1, 2, and 4, respectively, in the functions of the LabWindows/CVI X Property Library.***

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about `XInternAtom`.

DestroyXProperty

```
PropLibXErrType status = DestroyXProperty (PropertyHandleX property);
```

Purpose

Destroys X property information. You can use `DestroyXProperty` when you no longer need to access a property. This function frees memory that `CreateXProperty` allocates. The property handle cannot be used after you call `DestroyXProperty`.

LabWindows/CVI destroys all property information when the program terminates.

☞ **Note** ***You cannot destroy properties for which callbacks are installed. You must first call*** `UninstallXPropertyCallback` ***on the callbacks.***

Parameter

Input

Name	Type	Description
property	`PropertyHandleX`	Handle to the property information to destroy. This value must be one of the predefined property types or must be a value `CreateXPropType` returns.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidPropertyXErr`	`-4`	**property** argument is not a valid property. This argument must be the value `CreateXProperty` returns.
`PropertyInUseXErr`	`-10`	A property callback was installed with `InstallPropertyCallback` for this property. It is not possible to destroy properties for which callbacks are installed.

DestroyXPropType

```
PropLibXErrType status = DestroyXPropType (PropTypeHandleX propertyType);
```

Purpose

Destroys X property type. You can use `DestroyXPropType` when you no longer need a property type. This function frees memory that `CreateXPropType` allocates. The property type cannot be used after you call `DestroyXPropType`.

LabWindows/CVI destroys all property types when the program terminates.

Note ***You cannot destroy property types if there are properties that use them. You must first call*** `DestroyXProperty` ***on the properties.***

Parameter

Input

Name	Type	Description
propertyType	`PropertyHandleX`	Handle of the property type to destroy. This value must be one of the predefined property types or must be a value `CreateXPropType` returns.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidPropTypeXErr`	`-5`	**propertyType** argument is not a valid property type. This value must be one of the predefined property types, or must be a value `CreateXPropType` returns.
`PropTypeInUseXErr`	`-11`	A property that `CreateXProperty` created has this property type. It is not possible to destroy property types if there are properties that use them.

DisconnectFromXDisplay

```
PropLibXErrType status = DisconnectFromXDisplay (DisplayPtrX display);
```

Purpose

Disconnects from a remote X server. You can use `DisconnectFromXDisplay` to end access to a remote X server you connected to using `ConnectToXDisplay`. After you call `DisconnectFromXDisplay`, you can no longer access the remote X server.

Parameter

Input

Name	Type	Description
display	`DisplayPtrX`	A pointer to the display of the remote X server to disconnect. You must obtain the pointer from `ConnectToXDisplay`.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to the parameter.
`InvalidDisplayXErr`	`-2`	**display** argument is not a valid display. You must obtain a display pointer from `ConnectToXDisplay`.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about `XCloseDisplay`.

GetXPropErrorString

```
char *message = GetXPropErrorString (PropLibXErrType errorNum);
```

Purpose

Converts the error number an X Property Library function returns into a meaningful error message.

Parameter

Input

Name	Type	Description
errorNum	`PropLibXErrType`	Status that an X Property Library function returns.

Return Value

Name	Type	Description
message	string	Explanation of error.

GetXPropertyName

```
PropLibXErrType status = GetXPropertyName (PropertyHandleX property,
                                           char **propertyName);
```

Purpose

Gets a property name. This function returns a pointer to the name associated with the property handle.

Parameters

Input

Name	Type	Description
property	`PropertyHandleX`	Property handle for which to obtain the name. You must obtain this handle from `CreateXProperty`.

Output

Name	Type	Description
propertyName	character pointer	Pointer to the property name.

Caution ***The* propertyName *pointer points to memory allocated by*** `CreateXProperty`***. You must not attempt to free this pointer or to change its contents.***

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to the name parameter.
`InvalidPropertyXErr`	`-4`	**property** argument is not a valid property handle. You must obtain a property handle from `CreateXProperty`.

GetXPropertyType

```
PropLibXErrType status = GetXPropertyType (PropertyHandleX property,
                                           PropTypeHandleX *propertyType);
```

Purpose

Gets the type of a property.

This function returns a pointer to the type associated with the property handle.

Parameters

Input

Name	Type	Description
property	`PropertyHandleX`	Property handle for which to obtain the type. You must obtain this handle from `CreateXProperty`.

Output

Name	Type	Description
propertyType	`PropTypeHandleX`	Property type. Use `GetXPropTypeName`, `GetXPropTypeSize`, and `GetXPropTypeUnit` to get more information about the property type.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to the parameter.
`InvalidPropertyXErr`	`-4`	**property** argument is not a valid property handle. You must obtain a property handle from `CreateXProperty`.

GetXPropTypeName

```
PropLibXErrType status = GetXPropTypeName(PropTypeHandleX propertyType,
                                          char **typeName);
```

Purpose

Gets a property type name. This function returns the name associated with the property type.

Parameters

Input

Name	Type	Description
propertyType	`PropTypeHandleX`	Handle to property type for which to obtain the name. This value must be one of the predefined property types or a handle you obtain from `CreateXPropType`.

Output

Name	Type	Description
typeName	character pointer	Property type name.

Caution ***The* typeName *pointer points to memory*** `CreateXPropType` ***allocates. You must not attempt to free this pointer or to change its contents.***

Return Values

Constant Name	Value	Description
`NoXErr`	0	Function was successful.
`InvalidParamXErr`	-1	`NULL` was passed to the parameter.
`InvalidPropTypeXErr`	-5	**propertyType** argument is not a valid property type. You must use one of the predefined property types or obtain a property handle from `CreateXPropType`.

See Also

`CreateXPropType`

GetXPropTypeSize

```
PropLibXErrType status = GetXPropTypeSize (PropTypeHandleX propertyType,
                                           unsigned int *size);
```

Purpose

Gets a property type size. This function returns the size associated with the property type.

Parameters

Input

Name	Type	Description
propertyType	`PropTypeHandleX`	Handle to property type for which to obtain the size. This value must be one of the predefined property types or a handle you obtain from `CreateXPropType`.

Output

Name	Type	Description
size	unsigned integer	Size associated with the property type. The size is the number of bytes in a single property item.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to the parameter.
`InvalidPropTypeXErr`	`-5`	**propertyType** argument is not a valid property type. You must use one of the predefined property types or obtain a property handle from `CreateXPropType`.

See Also

`CreateXPropType`

GetXPropTypeUnit

```
PropLibXErrType status = GetXPropTypeUnit (PropTypeHandleX propertyType,
                          unsigned int *unit);
```

Purpose

Gets a property type unit.

`GetXPropTypeUnit` returns the unit associated with the property type.

Parameters

Input

Name	Type	Description
propertyType	`PropTypeHandleX`	Handle to property type for which to obtain the unit. This value must be one of the predefined property types or a handle you obtain from `CreateXPropType`.

Output

Name	Type	Description
unit	unsigned integer	**unit** associated with the property type. The unit is the number of bytes (1, 2 or 4) in the basic objects that make up a property item.

Return Values

Constant Name	Value	Description
NoXErr	0	Function was successful.
InvalidParamXErr	-1	NULL was passed to the parameter.
InvalidPropTypeXErr	-5	**propertyType** argument is not a valid property type. You must use one of the predefined property types or obtain a property handle from CreateXPropType.

See Also

CreateXPropType

GetXWindowPropertyItem

```
PropLibXErrType status = GetXWindowPropertyItem (DisplayPtrX display,
                          WindowX window, PropertyHandleX property,
                          void *propertyItem);
```

Purpose

Gets a single property item from a window.

`GetXWindowPropertyItem` obtains the value of the specified property on the window and copies a single item into the supplied buffer. When more than one item exists in the property value, `GetXWindowPropertyItem` obtains only the first one. This function does not change the property value.

If the property does not exist on the window, `GetXWindowPropertyItem` reports the `MissingPropertyXErr` error.

Use `GetXWindowPropertyValue` to get multiple property items.

Parameters

Input

Name	Type	Description
display	`DisplayPtrX`	Pointer to the display of the X server to which the window belongs.
window	`WindowX`	Window from which to obtain the property item.
property	`PropertyHandleX`	Handle of the property to obtain. You must obtain this handle from `CreateXProperty`.

Output

Name	Type	Description
propertyItem	generic pointer	Property item obtained from window.

Return Values

Constant Name	Value	Description
NoXErr	0	Function was successful.
InvalidParamXErr	-1	NULL was passed to one or more parameters.
InvalidDisplayXErr	-2	**display** argument is not a valid display. You must use the predefined value CVIXDisplay or obtain a display pointer from ConnectToXDisplay.
InvalidWindowXErr	-3	**window** argument is not a valid window.
InvalidPropertyXErr	-4	**property** argument is not a valid property handle. You must obtain a property handle from CreateXProperty.
TypeMismatchXErr	-12	Actual X type of the property value on the window does not match the type you specified for **property**.
UnitMismatchXErr	-13	Actual X format of the property value on the window does not match the unit you specified for **property**.
SizeMismatchXErr	-15	Number of bytes in the property value is not a multiple of the size you specified for **property**.
MissingPropertyXErr	-18	Property does not exist on the window.
InsuffMemXErr	-19	Insufficient memory to perform the operation.
GeneralXErr	-20	An Xlib function failed for an unknown reason.
BrokenConnectionXErr	-21	Connection to the X server was broken. This occurs if the remote server terminates.

Parameter Discussion

display must be the predefined value `CVIXDisplay` or must be a display pointer you obtain from `ConnectToXDisplay`. Use `CVIXDisplay` if the window is on the same display LabWindows/CVI uses.

For the **window** parameter, use `CVIXRootWindow` to access the default root window of the display LabWindows/CVI uses. Use `CVIXHiddenWindow` to access the hidden window associated with your application.

propertyItem must point to an object of the same size as the property item. You can get the size of the property item by calling the function `GetXPropertySize`.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about `XGetWindowProperty`.

GetXWindowPropertyValue

```
PropLibXErrType status = GetXWindowPropertyValue (DisplayPtrX display,
                          WindowX window, PropertyHandleX property,
                          unsigned int index,
                          unsigned int numberOfItemsRequested, int delete,
                          unsigned int *numberOfItemsReturned,
                          unsigned int *numberOfItemsRemaining,
                          void *propertyValue);
```

Purpose

Gets the value of a property on a window.

`GetXWindowPropertyValue` obtains the value of the specified property on the window and copies it into the supplied buffer.

☞ **Note** ***If the property does not exist on the window,*** `GetXWindowPropertyValue` ***does*** *not* ***report an error. Instead, it sets the number of items returned to*** `0`**.**

Parameters

Input

Name	Type	Description
display	`DisplayPtrX`	Pointer to the display of the X server to which the window belongs.
window	`WindowX`	Window from which to obtain the property value.
property	`PropertyHandleX`	Handle of the property to obtain. You must obtain this handle from `CreateXProperty`.
index	unsigned integer	Index into the property value where reading is to begin. Specify the number of property items to skip from the start of the property value.

Name	Type	Description
numberOfItemsRequested	unsigned integer	Number of property items to obtain from the window.
delete	integer	Flag that indicates whether to delete the property value from the window after it is obtained. Specify `1` to delete the portion of the property value that was obtained. Specify `0` to leave the property value as it is.

Output

Name	Type	Description
numberOfItemsReturned	unsigned integer	Number of property items that were obtained from the window.
numberOfItemsRemaining	unsigned integer	Number of property items on the window that were neither skipped nor obtained. Pass `NULL` for this parameter if you do not need this information.
propertyValue	generic pointer	Property value obtained from window. This parameter must point to an array of size *n*-by-*m* bytes, where *n* is the size of the property item, and *m* is the number of items requested.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to one or more parameters.

Constant Name	Value	Description
`InvalidDisplayXErr`	-2	**display** argument is not a valid display. You must use the predefined value `CVIXDisplay` or obtain a display pointer from `ConnectToXDisplay`.
`InvalidWindowXErr`	-3	**window** argument is not a valid window.
`InvalidPropertyError`	-4	**property** argument is not a valid property handle. You must obtain a handle from `CreateXProperty`.
`TypeMismatchXErr`	-12	Actual X type of the property value on the window does not match the type you specified for **property**.
`UnitMismatchXErr`	-13	Actual X format of the property value on the window does not match the unit you specified for **property**.
`InvalidIndexXErr`	-14	**index** you specified is larger than the actual number of property items on the window.
`SizeMismatchXErr`	-15	Number of bytes in the property value is not a multiple of the size you specified for **property**.
`InsuffMemXErr`	-19	Insufficient memory to perform the operation.
`GeneralXErr`	-20	An Xlib function failed for an unknown reason.
`BrokenConnectionXErr`	-21	Connection to the X server was broken. This occurs if the remote server terminates.

Parameter Discussion

display must be the predefined value `CVIXDisplay` or must be a display pointer you obtain from `ConnectToXDisplay`. Use `CVIXDisplay` if the window is on the same display LabWindows/CVI uses.

For the **window** parameter, use `CVIXRootWindow` to access the default root window of the display LabWindows/CVI uses. Use `CVIXHiddenWindow` to access the hidden window associated with your application.

numberOfItemsReturned will be less than or equal to the number of property items you requested. If the property does not exist on the window or no property value exists, this value is `0`. You must check this value to determine if any property items were read.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about `XGetWindowProperty`.

InstallXPropertyCallback

```
PropLibXErrType status = InstallXPropertyCallback (DisplayPtrX display,
                          const WindowX windowList[],
                          unsigned int numberOfWindows,
                          const PropertyHandleX propertyList[],
                          unsigned int numberOfProperties,
                          const void *callbackData,
                          PropertyCallbackTypeX *callbackFunction);
```

Purpose

Installs a property callback function.

The X Property Library calls the callback function whenever one of the specified properties on one of the specified windows changes in any way. If you install more than one function for the same property, the library calls the functions in the reverse order in which you installed them.

If you already installed the callback function, the window list and properties you specify in this call replace the ones you previously associated with the callback function.

Parameters

Input

Name	Type	Description
display	`DisplayPtrX`	Pointer to the display of the X server to which the window belongs.
windowList	`const WindowX []`	Array of windows on which the properties can exist.
numberOfWindows	unsigned integer	Number of windows in the window list; this value must be greater than `0`.
propertyList	`const Property CallbackTypeX []`	Array of handles to properties for which the library invokes the callback.
numberOfProperties	`unsigned integer`	Number of properties in the property list.

Name	Type	Description
callbackData	generic pointer	Pointer to data to pass to the callback function. LabWindows/CVI passes this value to the callback function as **userData**.
callbackFunction	`Property CallbackTypeX *`	Pointer to the function to call when the properties change.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to one or more parameters. The **numberOfWindows** argument is `0`.
`InvalidDisplayXErr`	`-2`	**display** argument is not a valid display. You must use the predefined value `CVIXDisplay` or obtain a display pointer from `ConnectToXDisplay`.
`InvalidWindowXErr`	`-3`	One or more of the windows in the **windowList** argument are not valid.
`InvalidPropertyXErr`	`-4`	One or more of the property handles in the **propertyList** argument are not valid. These properties must be values `CreateXProperty` returns.
`InsuffMemXErr`	`-19`	Insufficient memory to perform the operation.
`BrokenConnectionXErr`	`-21`	Connection to the X server was broken. This occurs if the remote server terminates.

Parameter Discussion

display must be the predefined value `CVIXDisplay` or must be a display pointer you obtain from `ConnectToXDisplay`. Use `CVIXDisplay` if the window is on the same display LabWindows/CVI uses.

To specify a single window named *`win`*, pass the expression `&`*`win`* for the **windowList** parameter and pass `1` for the **numberOfWindows**. Use `&CVIXRootWindow` to access the default root window of the display LabWindows/CVI uses. Use `&CVIXHiddenWindow` to specify the hidden window associated with your application.

If **numberOfProperties** is `0` or the **propertyList** value is `ANY_X_PROPERTY`, the callback function is called whenever any property changes on any of the windows in the **windowList**.

The values in the **propertyList** array must be handles you obtain from `CreateXProperty`.

To specify a single property named *`prop`*, pass the expression `&`*`prop`* for the **propertyList** parameter and pass `1` for **numberOfProperties**. If this value is `ANY_X_PROPERTY` or the **numberOfProperties** is `0`, the callback function is called whenever any property changes on any of the windows in the **windowList**.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about the `PropertyNotify` event.

PutXWindowPropertyItem

```
PropLibXErrType status = PutXWindowPropertyItem (DisplayPtrX display,
                          WindowX window, PropertyHandleX property,
                          void *propertyItem);
```

Purpose

Stores a single property item on a window. This value replaces any existing property value.

To store multiple property items, use `PutXWindowPropertyValue`.

Parameters

Input

Name	Type	Description
display	`DisplayPtrX`	Pointer to the display of the X server to which the window belongs.
window	`WindowX`	Window on which the property item is to be stored.
property	`PropertyHandleX`	Handle of the property to store. You must obtain this handle from `CreateXProperty`.
propertyItem	generic pointer	Property item to store on the window. This parameter must point to an object of the same size as a property item. You can get the property item size by calling `GetXPropertySize`.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to one or more parameters.

Constant Name	Value	Description
`InvalidDisplayXErr`	`-2`	**display** argument is not a valid display. You must use the predefined value `CVIXDisplay` or obtain a display pointer from `ConnectToXDisplay`.
`InvalidWindowXErr`	`-3`	**window** argument is not a valid window.
`InvalidPropertyXErr`	`-4`	**property** argument is not a valid property handle. You must obtain a property handle from `CreateXProperty`.
`InsuffMemXErr`	`-19`	Insufficient memory to perform the operation.
`GeneralXErr`	`-20`	An Xlib function failed for an unknown reason.
`BrokenConnectionXErr`	`-21`	Connection to the X server was broken. This occurs if the remote server terminates.

Parameter Discussion

display must be the predefined value `CVIXDisplay` or must be a display pointer you obtain from `ConnectToXDisplay`. Use `CVIXDisplay` if the window is on the same display LabWindows/CVI uses.

For the **window** parameter, use `CVIXRootWindow` to access the default root window of the display LabWindows/CVI uses. Use `CVIXHiddenWindow` to access the hidden window associated with your application.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about `XChangeProperty`.

PutXWindowPropertyValue

```
PropLibXErrType status = PutXWindowPropertyValue (DisplayPtrX display,
                          WindowX window, PropertyHandleX property,
                          unsigned int numberOfItems, int mode,
                          void *propertyValue);
```

Purpose

Stores the value of a property on a window.

To store a single property item, you can use `PutXWindowPropertyItem`.

Parameters

Input

Name	Type	Description
display	`DisplayPtrX`	Pointer to the display of the X server to which the window belongs.
window	`WindowX`	Window on which the property value is to be stored.
property	`PropertyHandleX`	Handle of the property to store. You must obtain this handle from `CreateXProperty`.
numberOfItems	unsigned integer	Number of property items to store on the window.
mode	integer	Mode in which property value is stored.
propertyValue	generic pointer	Property value to store on the window. This parameter must be an array of size *n*-by-*m* bytes, where *n* is the size of a property item, and *m* is the number of items to write.

Return Values

Constant Name	Value	Description
NoXErr	0	Function was successful.
InvalidParamXErr	-1	NULL was passed to one or more parameters; **mode** is not ReplaceXPropMode, PrependXPropMode or AppendXPropMode.
InvalidDisplayXErr	-2	**display** argument is not a valid display. You must use the predefined value CVIXDisplay or obtain a display pointer from ConnectToXDisplay.
InvalidWindowXErr	-3	**window** argument is not a valid window.
InvalidPropertyXErr	-4	**property** argument is not a valid property handle. You must obtain a property handle from CreateXProperty.
TypeMismatchXErr	-12	Actual X type of the property value on the window does not match the type you specified for **property**. This can occur only if you set **mode** to append or prepend.
UnitMismatchXErr	-13	Actual X format of the property value on the window does not match the unit you specified for **property**. This can occur only if you set **mode** to append or prepend.
OverflowXErr	-16	Arithmetic overflow occurred in calculations on the property item sizes and the number of items you specified.
InsuffMemXErr	-19	Insufficient memory to perform the operation.

Constant Name	Value	Description
GeneralXErr	-20	An Xlib function failed for an unknown reason.
BrokenConnectionXErr	-21	Connection to the X server was broken. This occurs if the remote server terminates.

Parameter Discussion

display must be the predefined value CVIXDisplay or must be a display pointer you obtain from ConnectToXDisplay. Use CVIXDisplay if the window is on the same display LabWindows/CVI uses.

For the **window** parameter, use CVIXRootWindow to access the default root window of the display LabWindows/CVI uses. Use CVIXHiddenWindow to access the hidden window associated with your application.

The following values are valid for the **mode** parameter:

ReplaceXPropMode—Replaces the existing property value with the new value.

PrependXPropMode—Adds the new property value to the beginning of the existing value.

AppendXPropMode—Adds the new property value to the end of the existing value.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about XChangeProperty.

RemoveXWindowProperty

```
PropLibXErrType status = RemoveXWindowProperty (DisplayPtrX display,
                          WindowX window, PropertyHandleX property);
```

Purpose

Deletes the property value and removes the property from the window.

Parameters

Input

Name	Type	Description
display	`DisplayPtrX`	Pointer to the display of the X server to which the window belongs.
window	`WindowX`	Window from which the property is to be removed.
property	`PropertyHandleX`	Handle of the property to remove. You must obtain this handle from `CreateXProperty`.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidParamXErr`	`-1`	`NULL` was passed to one or more parameters.
`InvalidDisplayXErr`	`-2`	**display** argument is not a valid display. You must use the predefined value `CVIXDisplay` or obtain a display pointer from `ConnectToXDisplay`.
`InvalidWindowXErr`	`-3`	**window** argument is not a valid window.
`InvalidPropertyXErr`	`-4`	**property** argument is not a valid property handle. You must obtain a property handle from `CreateXProperty`.

Constant Name	Value	Description
`InsuffMemXErr`	`-19`	Insufficient memory to perform the operation.
`BrokenConnectionXErr`	`-21`	Connection to the X server was broken. This occurs if the remote server terminates.

Parameter Discussion

display must be the predefined value `CVIXDisplay` or must be a display pointer you obtain from `ConnectToXDisplay`. Use `CVIXDisplay` if the window is on the same display LabWindows/CVI uses.

For the **window** parameter, use `CVIXRootWindow` to access the default root window of the display LabWindows/CVI uses. Use `CVIXHiddenWindow` to access the hidden window associated with your application.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about `XDeleteProperty`.

UninstallXPropertyCallback

```
PropLibXErrType status = UninstallXPropertyCallback
                          (PropertyCallbackTypeX *callbackFunction);
```

Purpose

Uninstalls a property callback function.

After a callback function is uninstalled, it is no longer called when properties change. LabWindows/CVI automatically uninstalls all property callback functions when the program terminates.

Note ***Although you cannot selectively uninstall certain properties or windows associated with a callback function, you can reinstall a callback function with a new set of windows and properties using*** `InstallXPropertyCallback`**.**

Parameter

Input

Name	Type	Description
callbackFunction	`Property CallbackTypeX *`	Function that was installed with `InstallXPropertyCallback`.

Return Values

Constant Name	Value	Description
`NoXErr`	`0`	Function was successful.
`InvalidCallbackXErr`	`-17`	Function you specified is not installed as a callback.

10

Easy I/O for DAQ Library

This chapter describes the functions in the Easy I/O for DAQ Library. The *Easy I/O for DAQ Library Function Overview* section contains general information about the functions and guidelines and restrictions you should know when using the Easy I/O for DAQ Library. The *Easy I/O for DAQ Function Reference* section contains an alphabetical list of function descriptions.

Easy I/O for DAQ Library Function Overview

The functions in the Easy I/O for DAQ Library make it easier to write simple DAQ programs than if you use the Data Acquisition Library.

This library implements a subset of the functionality of the Data Acquisition Library, but it does not use the same functions as the Data Acquisition Library. Read the following advantages and limitations to see if the Easy I/O for DAQ Library is appropriate for your application.

You must have NI-DAQ for PC compatibles installed to use the Easy I/O for DAQ Library, which has been tested using version 4.6.1 and later of NI-DAQ. It has not been tested using previous versions of NI-DAQ.

The `cvi\samples\easyio` directory includes sample programs for the Easy I/O for DAQ Library. The EASYIO section of `cvi\samples.doc` includes discussions of these sample programs.

☞ **Note** ***Do not mix calls to the Data Acquisition Library with similar types of calls to the Easy I/O for DAQ Library in the same application. For example, do not mix analog input calls to the Data Acquisition Library with analog input calls to the Easy I/O for DAQ Library in the same program.***

Advantages of Using the Easy I/O for DAQ Library

If you want to scan multiple analog input channels on an MIO board using the Data Acquisition Library, you have to programmatically build a channel list and a gain list before calling `SCAN_Op`.

The Easy I/O for DAQ functions accept a channel string and upper and lower input limit parameters so you can easily perform a scan in one step.

In the Data Acquisition Library, you might have to use `Lab_ISCAN_Op`, `SCAN_Op`, or `MDAQ_Start` depending on which DAQ device you use. Also, if you use SCXI, you must call a number of SCXI-specific functions before you actually acquire data.

The Easy I/O for DAQ functions are device independent, which means that you can use the same function on a Lab series board, an MIO board, an EISA-A2000, or an SCXI module.

Limitations of Using the Easy I/O for DAQ Library

The Easy I/O for DAQ Library currently works only with analog I/O, counter/timers, and simple digital I/O.

The library does not currently work with multirate scanning.

Easy I/O for DAQ Library Function Panels

The Easy I/O for DAQ Library function panels are grouped in the tree structure in Table 10-1 according to the types of operations they perform.

The first- and second-level headings in the function tree are names of the function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels. Each Easy I/O for DAQ function panel generates a function call.

Table 10-1. Functions in the Easy I/O for DAQ Library Function Tree

Class/Panel Name	Function Name
Analog Input	
AI Sample Channel	`AISampleChannel`
AI Sample Channels	`AISampleChannels`
AI Acquire Waveform(s)	`AIAcquireWaveforms`
AI Acq. Triggered Waveform(s)	`AIAcquireTriggeredWaveforms`
Asynchronous Acquisition	
AI Start Acquisition	`AIStartAcquisition`
AI Check Acquisition	`AICheckAcquisition`
AI Read Acquisition	`AIReadAcquisition`
AI Clear Acquisition	`AIClearAcquisition`
Plot Last Waveform(s) to Popup	`PlotLastAIWaveformsPopup`
Analog Output	
AO Update Channel	`AOUpdateChannel`
AO Update Channels	`AOUpdateChannels`
AO Generate Waveform(s)	`AOGenerateWaveforms`
AO Clear Waveform(s)	`AOClearWaveforms`
Digital Input/Output	
Read from Digital Line	`ReadFromDigitalLine`
Read from Digital Port	`ReadFromDigitalPort`

Table 10-1. Functions in the Easy I/O for DAQ Library Function Tree

Class/Panel Name	Function Name
Digital Input/Output (continued)	
Write to Digital Line	`WriteToDigitalLine`
Write to Digital Port	`WriteToDigitalPort`
Counter/Timer	
Counter Measure Frequency	`CounterMeasureFrequency`
Counter Event or Time Configure	`CounterEventOrTimeConfig`
Continuous Pulse Gen Configure	`ContinuousPulseGenConfig`
Delayed Pulse Gen Configure	`DelayedPulseGenConfig`
Frequency Divider Configure	`FrequencyDividerConfig`
Pulse Width or Period Meas Conf	`PulseWidthOrPeriodMeasConfig`
Counter Start	`CounterStart`
Counter Read	`CounterRead`
Counter Stop	`CounterStop`
I Counter Control	`ICounterControl`
Miscellaneous	
Get DAQ Error Description	`GetDAQErrorString`
Get Number of Channels	`GetNumChannels`
Get Channel Indices	`GetChannelIndices`
Get Channel Name from Index	`GetChannelNameFromIndex`
Get AI Limits of Channel	`GetAILimitsOfChannel`
Group by Channel	`GroupByChannel`
Set Multitasking Mode	`SetEasyIOMultitaskingMode`

Class Descriptions

- The Analog Input function class contains all the functions that perform A/D conversions.
- The Asynchronous Acquisition function class contains all the functions that perform asynchronous, or background, A/D conversions.
- The Analog Output function class contains all the functions that perform D/A conversions.
- The Digital Input/Output function class contains all the functions that perform digital input and output operations.
- The Counter/Timer function class contains all the functions that perform counting and timing operations.
- The Miscellaneous function class contains functions that do not fit into the other categories but are useful when you write programs using the Easy I/O for DAQ Library.

The online help with each panel contains specific information about operating each function panel.

Device Numbers

The first parameter to most of the Easy I/O for DAQ functions is the device number of the DAQ device you want to use for the given operation. After you have followed the installation and configuration instructions in Chapter 1, *Introduction to NI-DAQ*, of the *NI-DAQ User Manual for PC Compatibles*, the configuration utility displays the device number for each device you have installed in the system. You can use the configuration utility to verify your device numbers. To use multiple DAQ devices in one application, pass the appropriate device number to each function.

Channel String for Analog Input Functions

The second parameter to most of the analog input functions is the channel string that contains the analog input channels to sample.

Refer to Chapter 2, *Hardware Overview*, in your *NI-DAQ User Manual for PC Compatibles* to determine exactly what channels are valid for your hardware.

The following examples explain the syntax for the channel string in various cases:

- If you are using an MIO board, NEC-AI-16E-4, or NEC-AI-16XE-50, list the channels in the order in which they are to be read, as in the following example:

  ```
  "0,2,5"  /* reads channels 0, 2, and 5 in that order */
  "0:3"    /* reads channels 0 through 3 inclusive */
  ```

- If you are using AMUX-64T boards, you can address AMUX-64T channels when you attach one, two, or four AMUX-64T boards to a plug-in data acquisition board.

  Refer to Chapter 2, *Hardware Overview*, in your *NI-DAQ User Manual for PC Compatibles* to determine how AMUX-64T channels are multiplexed onto onboard channels.

  The onboard channel to which each block of four, eight, or 16 AMUX-64T channels are multiplexed and the scanning order of the AMUX-64T channels are fixed. To specify a range of AMUX-64T channels, you enter in the channel list the onboard channel into which the range is multiplexed. For example, if you have one AMUX-64T:

  ```
  "0"  /* reads channels 0 through 3 on each AMUX-64T board in that
  order */
  ```

  To sample a single AMUX-64T channel, you also must specify the number of the AMUX-64T board, as in the following example:

  ```
  "AM1!3" /* samples channel 3 on AMUX-64T board 1 */
  "AM4!8" /* samples channel 8 on AMUX-64T board 4 */
  ```

- If you are using a Lab-PC+, DAQCard-500/700/1200, DAQPad-1200, or PC-LPM-16, you can sample input channels only in descending order, and you must end with channel 0, such as `"3:0"`. If you are using a Lab-PC+ or 1200 product in differential mode, you must use even-numbered channels.

- If you are using a DAQPad-MIO-16XE-50, you can read the value of the cold junction compensation temperature sensor using `"cjtemp"` as the channel.
- If you are using SCXI, you can address SCXI channels when you attach one or more SCXI chassis to a plug-in data acquisition board. If you operate a module in parallel mode, you can select a SCXI channel by specifying the corresponding onboard channels or by using special channel syntax for SCXI. If you operate the modules in multiplexed mode, you must use the SCXI channel syntax. The following example describes the SCXI channel syntax:

  ```
  "OB1!SCx!MDy!a"  /* Channel a on the module in slot y of the chassis
  with ID x is multiplexed into onboard channel 1. */
  "OB0!SCx!MDy!a:b"  /* Channels a through b inclusive on the module
  in slot y of the chassis with ID x is multiplexed into onboard
  channel 0. */
  ```

SCXI channel ranges cannot cross module boundaries and must always increase in channel number.

The following examples of the SCXI channel syntax introduce the special SCXI channels:

- `"OB0!SCx!MDy!MTEMP"  /* The temperature sensor configured in MTEMP mode on the multiplexed module in slot y of the chassis with ID x. */`
- `"OB1!SCx!MDy!DTEMP"  /* The temperature sensor configured in DTEMP mode on the parallel module in slot y of the chassis with ID x. */`
- `"OB0!SCx!MDy!CALGND"  /* (SCXI-1100 and SCXI-1122 only) The grounded amplifier of the module in slot y of the chassis with ID x. */`
- `"OB0!SCx!MDy!SHUNT0"  /* (SCXI-1121, SCXI-1122 and SCXI-1321 only) Channel 0 of the module in slot y of the chassis with ID x, with the shunt resistor applied. */`
- `"OB0!SCx!MDy!SHUNT0:3" /* (SCXI-1121, SCXI-1122 and SCXI-1321 only) Channel 0 through 3 of the module in slot y of the chassis with ID x, with the shunt resistors applied at each channel. */`

Command Strings

You can use command strings within the channel string to set per-channel limits and an interchannel sample rate. For example,

```
"cmd hi 10.0 low -10.0; 7:4; cmd hi 5.0 low -5.0; 3:0"
```

specifies that channels `7` through `4` should be scanned with limits of `+/- 10.0` volts and channels `3` through `0` should be scanned with limits of `+/- 5.0` volts. As you view the channel string from left to right, each high/low limit command applies to the channels that

follow it until the next high/low limit command is encountered. The **highLimit** and **lowLimit** parameters to `AISampleChannels` are the initial high/low limits, which apply to channels in the channel string to the left of the first high/low limit command.

The channel string

```
"cmd interChannelRate 1000.0; 0:3"
```

specifies that channels `0` through `3` should be sampled at 1,000.0 Hz. In other words, there should be 1/1,000.0 = 1 ms of delay between each channel. If you do not set an interchannel sample rate, the channels are sampled as fast as possible for your hardware to achieve pseudo-simultaneous scanning.

The following guide describes the syntax for the command string:

- Items enclosed in `[]` are optional.
- `<number>` is an integer or real number.
- `<LF>` is a linefeed character.
- `;|<LF>` means you can use either `;` or `<LF>` to separate command strings from channel strings.
- You can use `!` as an optional command separator.
- Spaces are optional.

The following example specifies the syntax for the initial command string that appears before any channel:

```
"cmd [interChannelRate <number>[!]] [hi <number> [!]low
    <number>[!]];|<LF>"
```

The following example specifies the syntax for command strings that appear after any channel:

```
";|<LF> cmd hi <number>[!] low <number>[!] ;|<LF>"
```

Channel String for Analog Output Functions

The second parameter to most of the analog output functions is the channel string that contains the analog output channels to drive.

Refer to the chapter specific to your DAQ device in the *DAQ Hardware Overview Guide* to determine what channels are valid for your hardware. The document is an Adobe Acrobat file, `daqhwov.pdf`, that you can view on screen or print. `daqhwov.pdf` is part of a set of `.pdf` files that National Instruments includes with every DAQ device the company sells.

The following examples explain the syntax for the channel string:

- If you are using a DAQ device without SCXI, list the channels to drive, as in the following example:

  ```
  "0,2,5"  /* drives channels 0, 2, and 5 */
  "0:3"    /* drives channels 0 through 3 inclusive */
  ```

- If you are using SCXI, you can address SCXI channels when you attach one or more SCXI chassis to a plug-in data acquisition board by using the following SCXI channel syntax:

  ```
  "SCx!MDy!a"   /* Channel a on the module in slot y of the chassis
  with ID x. */
  "SCx!MDy!a:b"  /* Channels a through b inclusive on the module in
  slot y of the chassis with ID x. */
  ```

  SCXI channel ranges cannot cross module boundaries and must always increase in channel number.

Valid Counters for the Counter/Timer Functions

The second parameter to most of the counter/timer functions is the counter the functions use for the operation. The valid counters you can use depends on your hardware, as shown in Table 10-2.

Table 10-2. Valid Counters

Device Type	Valid Counters
DAQ-STC Devices	`0` and `1`
Am9513 MIO boards	`1`, `2`, and `5`
PC-TIO-10	`1`–`10`
EISA-A2000	`2`

Easy I/O for DAQ Function Reference

This section describes each function in the Easy I/O for DAQ Library in alphabetical order.

AIAcquireTriggeredWaveforms

```
short error = AIAcquireTriggeredWaveforms (short device,
                          char channelString[], long numberOfScans,
                          double scansPerSecond, double highLimit,
                          double lowLimit, double *actualScanRate,
                          unsigned short triggerType,
                          unsigned short edgeSlope, double triggerLevel,
                          char triggerSource[], long pretriggerScans,
                          double timeLimitsec, short fillMode,
                          double waveforms[]);
```

Purpose

Performs a timed acquisition of voltage data from the analog channels you specify in **channelString**. The acquisition does not start until the trigger conditions are satisfied.

If you have an E Series DAQ device, you can select Equivalent Time Sampling for **triggerType** to sample repetitive waveforms at up to 20 MHz. Refer to the following *Parameter Discussion* section for more information.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
channelString	string	Analog input channels to sample.
numberOfScans	long integer	Number of scans to acquire. One scan involves sampling every channel in **channelString** once.
scansPerSecond	double	Number of scans performed per second. The function samples each channel at this rate.
highLimit	double	Maximum value to measure.
lowLimit	double	Minimum value to measure.
triggerType	unsigned short integer	Trigger type.
edgeSlope	unsigned short integer	Edge/slope condition for triggering.
triggerLevel	double	Value at which the trigger is to occur.

Name	Type	Description
triggerSource	string	Specifies which channel is the trigger source.
pretriggerScans	long integer	Specifies the number of scans to retrieve before the trigger point.
timeLimitsec	double	Maximum length of time in seconds to wait for the data.
fillMode	short integer	Specifies whether the **waveforms** array is in `GROUP_BY_CHANNEL` or `GROUP_BY_SCAN` mode.

Output

Name	Type	Description
actualScanRate	double	Actual scan rate. The actual scan rate might differ slightly from the scan rate you specified, given the limitations of your particular DAQ device.
waveforms	double array	Array that contains the values acquired on the channels you specify in **channelString**.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the analog input channel to sample. Refer to the *Channel String for Analog Input Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

triggerType is the trigger type. Table 10-3 lists trigger types.

Table 10-3. Trigger Types

Trigger Type	Constant Name
Hardware Analog Trigger	`HW_ANALOG_TRIGGER`
Digital Trigger A	`DIGITAL_TRIGGER_A`
Digital Triggers A & B	`DIGITAL_TRIGGER_AB`
Scan Clock Gating	`SCAN_CLOCK_GATING`
Software Analog Trigger	`SW_ANALOG_TRIGGER`
Equivalent Time Sampling	`ETS_TRIGGER`

If you choose Hardware or Software Analog Trigger, `AIAcquireTriggeredWaveforms` retrieves data after the analog triggering parameters are satisfied. Be sure that **triggerSource** is one of the channels the channel string lists. Hardware triggering is more accurate than software triggering, but it is not available on all boards.

If you choose Digital Trigger A, the trigger starts the acquisition if **pretriggerScans** is `0`. For the MIO-16, connect the digital trigger signal to the STARTTRIG input. If **pretriggerScans** is greater than `0`, the trigger stops the acquisition after all posttrigger data is acquired. For the MIO-16, connect the digital trigger signal to the STOPTRIG input.

If you choose Digital Trigger A & B, **pretriggerScans** must be greater than `0`. A digital trigger starts the acquisition and a digital trigger stops the acquisition after all posttrigger data is acquired. For the MIO-16, the STARTTRIG input starts the acquisition and the STOPTRIG input stops the acquisition.

If you choose Scan Clock Gating, an external signal gates the scan clock on and off. If the scan clock gate becomes FALSE, the current scan completes, and the scan clock ceases operation. When the scan clock gate becomes TRUE, the scan clock immediately begins operation again.

If you choose Equivalent Time Sampling, you use the Equivalent Time Sampling technique on an E Series DAQ device to achieve an effective acquisition rate of up to 20 MHz, with the following conditions:

- The signal that you measure must be a periodic waveform.
- The trigger conditions must be satisfied or `AIAcquireTriggeredWaveforms` times out.

Equivalent Time Sampling is the process of taking A/D conversions from a periodic waveform at special points in time such that when the A/D conversions are placed side by side, they represent the original waveform as if it were sampled at a high frequency.

For example, if the A/D conversions, represented by `x`'s, on the waveform shown in Figure 10-1 are placed side by side, they represent one cycle of the waveform.

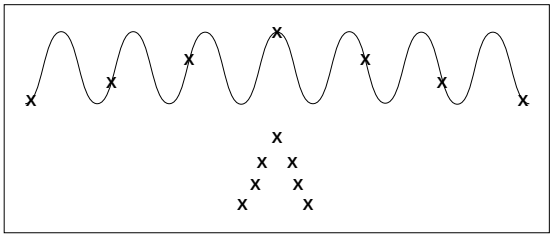

Figure 10-1. One Cycle of a Waveform

`AIAcquireTriggeredWaveforms` accomplishes Equivalent Time Sampling as follows:

- You set a hardware analog trigger condition for measuring your waveform using **edgeSlope**, **triggerLevel**, and **triggerSource**.
- Whenever a hardware analog trigger occurs, `AIAcquireTriggeredWaveforms` strobes the internal ATCOUT signal.
- `AIAcquireTriggeredWaveforms` internally routes the ATCOUT signal to the gate of GPCTR0, which is configured to generate a pulse each time it receives a rising edge at its gate input.
- The function then internally routes the output of GPCTR0 to the data acquisition sample clock to control the A/D conversion rate.
- You can achieve a very high effective scan rate through a pre-pulse delay that is programmed into GPCTR0. This delay automatically increments before each GPCTR0 pulse so that the A/D conversions occur at slightly larger intervals from the trigger condition as trigger conditions occur over time.
- Because the waveform being measured is periodic, A/D conversions that are at particular intervals from trigger conditions over time can look the same as A/D conversions at particular intervals from one unique trigger point in time, as shown in Figure 10-2 and Figure 10-3.

In Figure 10-2:

`t`*n* = nth trigger condition
`d`*n* = delay between the nth trigger and the nth conversion
`x` = an A/D conversion
`---` = the trigger level

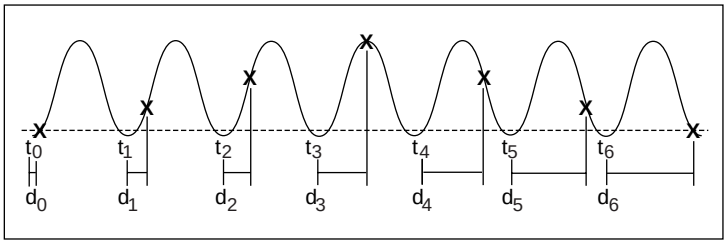

Figure 10-2. Converting a Signal at Periodic Intervals

When the A/D conversions are placed side by side, they represent the original waveform as if it were sampled at a high frequency.

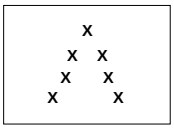

Figure 10-3. Resulting Waveform That Resembles Original Waveform

edgeSlope specifies whether the trigger occurs when the trigger signal value is leading, `POSITIVE_SLOPE`, or trailing, `NEGATIVE_SLOPE`.

triggerLevel specifies the value at which the trigger is to occur. **triggerLevel** is valid only when **triggerType** is hardware or software analog trigger.

triggerSource specifies which channel the trigger source is. **triggerSource** must be one of the channels **channelString** lists. If you pass `""` or `NULL`, the function uses the first channel in **channelString** as **triggerSource**. **triggerSource** is valid only when **triggerType** is hardware or software analog trigger.

timeLimitsec is the maximum length of time in seconds to wait for the data. If the time you set expires, `AIAcquireTriggeredWaveforms` returns a timeout error (`timeOutErr = -10800`). A value of `-2.0` disables the time limit.

Caution ***Disabling the time limit leaves your program in a suspended state until the trigger condition occurs.***

A value of `-1.0` (default) for **timeLimitsec** lets the function calculate the timeout based on the acquisition rate and number of scans you request.

fillMode specifies whether the **waveforms** array is grouped by channels or grouped by scans. Consider the following examples:

- If you scan channels A through C and **numberOfScans** is `5`, the possible fill modes are as follows:

  `GROUP_BY_CHANNEL`

  $$\underbrace{\texttt{A1 A2 A3 A4 A5}}\quad\underbrace{\texttt{B1 B2 B3 B4 B5}}\quad\underbrace{\texttt{C1 C2 C3 C4 C5}}$$

  `GROUP_BY_SCAN`

  $$\underbrace{\texttt{A1 B1 C1}}\quad\underbrace{\texttt{A2 B2 C2}}\quad\underbrace{\texttt{A3 B3 C3}}\quad\underbrace{\texttt{A4 B4 C4}}\quad\underbrace{\texttt{A5 B5 C5}}$$

- If you pass the array to a graph, acquire the data grouped by channel.
- If you pass the array to a strip chart, acquire the data grouped by scan.
- You can acquire the data grouped by scan and later reorder it to be grouped by channel using `GroupByChannel`.

waveforms is an array that contains the values acquired on the channels you specify in the **channelString**. `AIAcquireTriggeredWaveforms` places the acquired values into the array in the order **fillMode** specifies. This array must be declared as large as (number of channels) × (**numberOfScans**). You can determine the number of channels using `GetNumChannels`.

If you set **highLimit** and **lowLimit** to `0.0`, this function uses the default limits, which are defined as follows:

- For channels you configure in the DAQ Channel Wizard, the default limits are set in the Physical Quantity section of the DAQ Channel Wizard.
- For hardware channels, the default limits are set in the AI section of NI-DAQ Configuration Utility.

AIAcquireWaveforms

```
short error = AIAcquireWaveforms (short device, char channelString[],
                                  long numberOfScans, double scansPerSecond,
                                  double highLimit, double lowLimit,
                                  double *actualScanRate, short fillMode,
                                  double waveforms[]);
```

Purpose

Performs a timed acquisition of voltage data from the analog channels you specify in **channelString**.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
channelString	string	Analog input channels to sample.
numberOfScans	long integer	Number of scans to acquire. One scan involves sampling every channel in **channelString** once.
scansPerSecond	double	Number of scans performed per second. The function samples each channel at this rate.
highLimit	double	Maximum value to measure.
lowLimit	double	Minimum value to measure.
fillMode	short integer	Specifies whether the **waveforms** array is in `GROUP_BY_CHANNEL` or `GROUP_BY_SCAN` mode.

Output

Name	Type	Description
actualScanRate	double	Actual scan rate. The actual scan rate might differ slightly from the scan rate you specified, given the limitations of your particular DAQ device.
waveforms	double array	Array that contains the values acquired on the channels you specify in the **channelString**.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the analog input channel to sample. Refer to the *Channel String for Analog Input Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

fillMode specifies whether the **waveforms** array is grouped by channels or grouped by scans. Consider the following examples:

- If you scan channels A through C and **numberOfScans** is `5`, the possible fill modes are as follows:

  `GROUP_BY_CHANNEL`

  $$\underbrace{\texttt{A1 A2 A3 A4 A5}}\quad\underbrace{\texttt{B1 B2 B3 B4 B5}}\quad\underbrace{\texttt{C1 C2 C3 C4 C5}}$$

  `GROUP_BY_SCAN`

  $$\underbrace{\texttt{A1 B1 C1}}\quad\underbrace{\texttt{A2 B2 C2}}\quad\underbrace{\texttt{A3 B3 C3}}\quad\underbrace{\texttt{A4 B4 C4}}\quad\underbrace{\texttt{A5 B5 C5}}$$

- If you pass the array to a graph, acquire the data grouped by channel.
- If you pass the array to a strip chart, acquire the data grouped by scan.
- You can acquire the data grouped by scan and later reorder it to be grouped by channel using `GroupByChannel`.

waveforms is an array that contains the values acquired on the channels you specify in the **channelString**. `AIAcquireWaveforms` places the acquired values into the array in the order **fillMode** specifies.

This array must be declared as large as (number of channels) × (**numberOfScans**) . You can determine the number of channels using `GetNumChannels`.

If you set **highLimit** and **lowLimit** to `0.0`, this function uses the default limits, which are defined as follows:

- For channels you configure in the DAQ Channel Wizard, the default limits are set in the Physical Quantity section of the DAQ Channel Wizard.
- For hardware channels, the default limits are set in the AI section of NI-DAQ Configuration Utility.

AICheckAcquisition

```
short error = AICheckAcquisition (unsigned long taskID,
                                  unsigned long *scanBacklog);
```

Purpose

Determines the backlog of scans that have been acquired into the circular buffer but that you have not read using `AIReadAcquisition`.

If you call `AIReadAcquisition` with read mode set to `LATEST_MODE`, `AICheckAcquisition` resets **scanBacklog** to zero.

Parameters

Input

Name	Type	Description
taskID	unsigned long integer	Task ID that `AIStartAcquisition` returns.

Output

Name	Type	Description
scanBacklog	unsigned long integer	Returns the backlog of scans that have been acquired into the circular buffer but that you have not read using `AIReadAcquisition`.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

AIClearAcquisition

```
short error = AIClearAcquisition (unsigned long taskID);
```

Purpose

Clears the current asynchronous acquisition that `AIStartAcquisition` started.

Parameter

Input

Name	Type	Description
taskID	unsigned long integer	Task ID that `AIStartAcquisition` returns.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

AIReadAcquisition

```
short error = AIReadAcquisition (unsigned long taskID, long scanstoRead,
                                 unsigned short readMode,
                                 unsigned long *scanBacklog,
                                 short fillMode, double waveforms[]);
```

Purpose

Reads the specified number of scans from the internal circular buffer established by `AIStartAcquisition`.

If the specified number of scans is not available in the buffer, the function waits until the scans are available. You can call `AICheckAcquisition` before calling `AIReadAcquisition` to determine how many scans are available.

Parameters

Input

Name	Type	Description
taskID	unsigned long integer	Task ID that `AIStartAcquisition` returns.
scanstoRead	long integer	Number of scans to read from the internal circular buffer.
readMode	unsigned short integer	Specifies whether to read scans from the circular buffer in `CONSECUTIVE_MODE` or `LATEST_MODE`.
fillMode	short integer	Specifies whether the **waveforms** array is in `GROUP_BY_CHANNEL` or `GROUP_BY_SCAN` mode.

Output

Name	Type	Description
scanBacklog	unsigned long integer	Returns the backlog of scans that have been acquired into the circular buffer but that you have not read using `AIReadAcquisition`.
waveforms	double array	Array that contains the values acquired on the channels you specify in the **channelString**.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

readMode specifies whether `AIReadAcquisition` reads scans from the circular buffer in `CONSECUTIVE_MODE` or `LATEST_MODE`. In `CONSECUTIVE_MODE`, the function reads scans from the internal circular buffer starting from the last scan that you read. Using this mode, you are guaranteed not to lose data unless an error occurs. In `LATEST_MODE`, `AIReadAcquisition` reads the most recently acquired *n* scans from the internal circular buffer, where *n* is **scanstoRead**. Calling `AIReadAcquisition` in this mode resets the **scanBacklog** to zero.

scanBacklog returns the backlog of scans that have been acquired into the circular buffer but that you have not read using `AIReadAcquisition`. You can call `AICheckAcquisition` to determine the scan backlog before you call `AIReadAcquisition`.

waveforms is an array that contains the values acquired on the channels you specify in the **channelString**. `AIReadAcquisition` places the acquired values into the array in the order **fillMode** specifies.

This array must be declared as large as (number of channels) × (**scanstoRead**). You can determine the number of channels using `GetNumChannels`.

AISampleChannel

```
short error = AISampleChannel (short device, char singleChannel[],
                               double highLimit, double lowLimit,
                               double *sample);
```

Purpose

Acquires a single voltage from a single analog input channel.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by the NI-DAQ configuration utility.
singleChannel	string	Analog input channel to sample.
highLimit	double	Maximum value to measure.
lowLimit	double	Minimum value to measure.

Output

Name	Type	Description
sample	double (passed by reference)	The measured value.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

singleChannel is the analog input channel to sample. Refer to the *Channel String for Analog Input Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

If you set **highLimit** and **lowLimit** to `0.0`, this function uses the default limits, which are defined as follows:

- For channels you configure in the DAQ Channel Wizard, the default limits are set in the Physical Quantity section of the DAQ Channel Wizard.
- For hardware channels, the default limits are set in the AI section of NI-DAQ Configuration Utility.

AISampleChannels

```
short error = AISampleChannels (short device, char channelString[],
                                double highLimit, double lowLimit,
                                double sampleArray[]);
```

Purpose

Performs a single scan on a set of analog input channels.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by the NI-DAQ configuration utility.
channelString	string	Analog input channels to sample.
highLimit	double	Maximum value to measure.
lowLimit	double	Minimum value to measure.

Output

Name	Type	Description
sampleArray	double array	Array that contains the values acquired on the channels you specify in **channelString**.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the Analog input channels to sample. Refer to the *Channel String for Analog Input Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

sampleArray is an array that contains the values acquired on the channels you specify in **channelString**. `AISampleChannels` places the acquired values into the array in the order you specify in **channelString**. This array must be declared as large as the number of channels you specify in **channelString**.

You can use `GetNumChannels` to determine the number of channels.

If you set **highLimit** and **lowLimit** to `0.0`, this function uses the default limits, which are defined as follows:

- For channels you configure in the DAQ Channel Wizard, the default limits are set in the Physical Quantity section of the DAQ Channel Wizard.
- For hardware channels, the default limits are set in the AI section of NI-DAQ Configuration Utility.

AIStartAcquisition

```
short error = AIStartAcquisition (short device, char channelString[],
                                  int bufferSize, double scansPerSecond,
                                  double highLimit, double lowLimit,
                                  double *actualScanRate, unsigned long *taskID);
```

Purpose

Starts a continuous asynchronous acquisition on the analog input channels you specify in **channelString**. `AIStartAcquisition` acquires data into an internal circular buffer. Use `AIReadAcquisition` to retrieve scans from the internal buffer.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
channelString	string	Analog input channels to sample.
bufferSize	integer	Size of the internal circular buffer in scans.
scansPerSecond	double	Number of scans performed per second. The function samples each channel at this rate.
highLimit	double	Maximum value to measure.
lowLimit	double	Minimum value to measure.

Output

Name	Type	Description
actualScanRate	double	Actual scan rate. The actual scan rate might differ slightly from the scan rate you specified, given the limitations of your particular DAQ device.
taskID	unsigned long integer	Identifier for the asynchronous acquisition.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the Analog input channels to sample. Refer to the *Channel String for Analog Input Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

taskID is an identifier for the asynchronous acquisition that you must pass to `AICheckAcquisition`, `AIReadAcquisition`, and `AIClearAcquisition`.

If you set **highLimit** and **lowLimit** to `0.0`, this function uses the default limits, which are defined as follows:

- For channels you configure in the DAQ Channel Wizard, the default limits are set in the Physical Quantity section of the DAQ Channel Wizard.
- For hardware channels, the default limits are set in the AI section of NI-DAQ Configuration Utility.

AOClearWaveforms

```
short error = AOClearWaveforms (unsigned long taskID);
```

Purpose

Clears the waveforms `AOGenerateWaveforms` generates when you pass `0` for **Iterations**.

Parameter

Input

Name	Type	Description
taskID	unsigned long integer	Task ID `AOGenerateWaveforms` returns.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

AOGenerateWaveforms

```
short error = AOGenerateWaveforms (short device, char channelString[],
                                   double updatesPerSecond, int updatesPerChannel,
                                   int iterations, double waveforms[],
                                   unsigned long *taskID);
```

Purpose

Generates a timed waveform of voltage data on the analog output channels you specify in **channelString**.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by the NI-DAQ configuration utility.
channelString	string	The analog output channels to apply the voltages to.
updatesPerSecond	double	Number of updates to perform per second. The function updates each channel at this rate.
updatesPerChannel	integer	Number of D/A conversions that compose a waveform for a particular channel.
iterations	integer	Number of waveform iterations to perform before the operation is complete; `0` = continuous.

Output

Name	Type	Description
waveforms	double array	Voltages to apply to the channels you specify in **channelString**.
taskID	unsigned long integer	Identifier for the waveform generation. If you pass `0` as the **iterations** parameter, you need to pass **taskID** to `AOClearWaveforms` to clear the waveform generation.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the analog output channels to apply the voltages to. Refer to the *Channel String for Analog Output Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

updatesPerChannel is the number of D/A conversions that compose a waveform for a particular channel. If **updatesPerChannel** is `10`, each waveform is composed of 10 elements from the **waveforms** array.

iterations is the number of waveform iterations performed before the operation is complete. If you pass `0`, `AOGenerateWaveforms` generates the waveform(s) continuously and you need to call `AOClearWaveforms` to clear waveform generation.

waveforms is the array that contains the voltages to apply to the channels you specify in **channelString**. The voltages are applied to the analog output channels in the order you specify in **channelString**. For example, if **channelString** is

```
"0:3,5",
```

the array contains the voltages in the following order:

```
waveforms[0]     /*  the 1st update on channel 0 */
waveforms[1]     /*  the 1st update on channel 1 */
waveforms[2]     /*  the 1st update on channel 2 */
waveforms[3]     /*  the 1st update on channel 3 */
waveforms[4]     /*  the 1st update on channel 5 */
waveforms[5]     /*  the 2nd update on channel 0 */
waveforms[6]     /*  the 2nd update on channel 1 */
waveforms[7]     /*  the 2nd update on channel 2 */
waveforms[8]     /*  the 2nd update on channel 3 */
waveforms[9]     /*  the 2nd update on channel 5 */
.
.
.
waveforms[n-5]   /*  the last update on channel 0 */
waveforms[n-4]   /*  the last update on channel 1 */
waveforms[n-3]   /*  the last update on channel 2 */
waveforms[n-2]   /*  the last update on channel 3 */
waveforms[n-1]   /*  the last update on channel 5 */
```

AOUpdateChannel

```
short error = AOUpdateChannel (short device, char singleChannel[],
                               double voltage);
```

Purpose

Applies the voltage you specify to a single analog output channel.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
singleChannel	string	Analog output channel to which the voltage is applied.
voltage	double	Voltage applied to the analog output channel.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

singleChannel is the analog output channel to which the voltage is applied. Refer to the *Channel String for Analog Output Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

AOUpdateChannels

```
short AOUpdateChannels (short device, char channelString[],
                        double voltageArray[]);
```

Purpose

Applies the voltages you specify in **voltageArray** to the analog output channel you specify in **channelString**.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
channelString	string	Analog output channels to which the voltages are applied.
voltageArray	double array	Voltages applied to the analog output channels you specify.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the analog output channels to which the voltages are applied. Refer to the *Channel String for Analog Output Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

voltageArray is
the voltages that are applied to the analog output channels you specify. This array contains the voltages to apply to the analog output channels in the order that you specify in **channelString**. For example, if **channelString** contains:

```
"0,1,3"
```

then:

```
voltage[0] = 1.2;   /* 1.2 volts applied to channel 0  */
voltage[1] = 2.4;   /* 2.4 volts applied to channel 1  */
voltage[2] = 3.6;   /* 3.6 volts applied to channel 3  */
```

ContinuousPulseGenConfig

```
short error = ContinuousPulseGenConfig (short device, char counter[],
                            double frequency, double dutyCycle,
                            unsigned short gateMode,
                            unsigned short pulsePolarity,
                            double *actualFrequency,
                            double *actualDutyCycle,
                            unsigned long *taskID);
```

Purpose

Configures a counter to generate a continuous Transistor-Transistor Logic (TTL) pulse train on its OUT pin.

`ContinuousPulseGenConfig` creates the signal by repeatedly decrementing the counter twice, first for the delay to the pulse, phase 1, then for the pulse itself, phase 2. The function selects the highest resolution timebase to achieve the desired characteristics.

You can gate or trigger the operation with a signal on the counter GATE pin. Call `CounterStart` to start the operation or to enable the gate or trigger action.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
counter	string	Counter to use for the counting operation.
frequency	double	Desired repetition rate of the continuous pulse train.
dutyCycle	double	Desired ratio of the duration of the pulse phase, phase 2, to the period, (phase 1 + phase 2).
gateMode	unsigned short integer	Specifies how to use the signal on the counter GATE pin.
pulsePolarity	unsigned short integer	Polarity of phase 2 of each cycle.

Output

Name	Type	Description
actualFrequency	double	Achieved frequency based on the resolution and range of your hardware.
actualDutyCycle	double	Achieved duty cycle based on the resolution and range of your hardware.
taskID	unsigned long integer	Reference number assigned to this operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

counter is the counter to use for the counting operation. For valid counters, refer to Table 10-2 in the *Valid Counters for the Counter/Timer Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter.

dutyCycle is the desired ratio of the duration of the pulse phase, phase 2, to the period, phase 1 + phase 2. The default of `0.5` generates a square wave.

- If **dutyCycle** = `0.0`, `ContinuousPulseGenConfig` computes the closest achievable duty cycle using a minimum pulse phase, phase 2, of three timebase cycles.
- If **dutyCycle** = `1.0`, `ContinuousPulseGenConfig` computes the closest achievable duty cycle using a minimum delay phase, phase 1, of three timebase cycles.
- A duty cycle very close to `0.0` or `1.0` might not be possible.

gateMode specifies how to use the signal on the counter GATE pin. You can use the following options:

- `UNGATED_SOFTWARE_START`—Ignore the gate signal and start when you call `CounterStart`.
- `COUNT_WHILE_GATE_HIGH`—Count while the gate signal is TTL high after you call `CounterStart`.
- `COUNT_WHILE_GATE_LOW`—Count while the gate signal is TTL low after you call `CounterStart`.

- `START_COUNTING_ON_RISING_EDGE`—Start counting on the rising edge of the TTL gate signal after you call `CounterStart`.
- `START_COUNTING_ON_FALLING_EDGE`—Start counting on the falling edge of the TTL gate signal after you call `CounterStart`.

pulsePolarity is the polarity of phase 2 of each cycle. You can use the following options:

- `POSITIVE_POLARITY`—The delay, phase 1, is a low TTL level, and the pulse, phase 2, is a high level.
- `NEGATIVE_POLARITY`—The delay, phase 1, is a high TTL level, and the pulse, phase 2, is a low level.

CounterEventOrTimeConfig

```
short error = CounterEventOrTimeConfig (short device, char counter[],
                            unsigned short counterSize,
                            double sourceTimebase,
                            unsigned short countLimitAction,
                            short sourceEdge, unsigned short gateMode,
                            unsigned long *taskID);
```

Purpose

Configures one or two counters to count edges in the signal on the counter SOURCE pin you specify or the number of cycles of an internal timebase signal you specify.

When you use `CounterEventOrTimeConfig` with the internal timebase and in conjunction with `CounterStart` and `CounterRead`, your program can make more precise timing measurements than with `Timer`.

You can gate or trigger the operation with a signal on the counter GATE pin. Call `CounterStart` to start the function or to enable the gate or trigger action.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
counter	string	Counter to use for the counting operation.
counterSize	unsigned short integer	Determines the size of the counter used to perform the operation.
sourceTimebase	double	Use `USE_COUNTER_SOURCE` to count TTL edges at the counter SOURCE pin or supply a valid internal timebase frequency to count the TTL edges of an internal clock.
countLimitAction	unsigned short integer	Action to take when the counter reaches terminal count.
sourceEdge	short integer	Edge of the counter source or timebase signal on which it increments.
gateMode	unsigned short integer	Specifies how to use the signal on the counter GATE pin.

Output

Name	Type	Description
taskID	unsigned long integer	Reference number assigned to this counter operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

counter is the counter to use for the counting operation. For valid counters, refer to Table 10-2 in the *Valid Counters for the Counter/Timer Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter.

counterSize determines the size of the counter `CounterEventOrTimeConfig` uses to perform the operation. For a device with DAQ-STC counters, **counterSize** must be `ONE_COUNTER` (24-bit). For a device with Am9513 counters, **counterSize** can be `ONE_COUNTER` (16-bit) or `TWO_COUNTERS` (32-bit). If you use `TWO_COUNTERS`, **counter**+1 is cascaded with the counter you specify. Table 10-4 defines **counter**+1.

Table 10-4. Definition of Am9513: Counter+1

counter	counter+1
1	2
2	3
3	4
4	5
5	1
6	7
7	8
8	9
9	10
10	6

sourceTimebase determines whether the counter uses its SOURCE pin or an internal timebase as its signal source. Pass `USE_COUNTER_SOURCE` to count TTL edges at the counter SOURCE pin or pass a valid internal timebase frequency to count the TTL edges of an internal clock.

Table 10-5 shows valid internal timebase frequencies are:

Table 10-5. Valid Internal Timebase Frequencies

Frequency	Chip Type on DAQ Board
1,000,000	Am9513
100,000	Am9513
10,000	Am9513
1,000	Am9513
100	Am9513
20,000,000	DAQ-STC
100,000	DAQ-STC

countLimitAction is the action to take when the counter reaches terminal count and accepts the following attributes:

- `COUNT_UNTIL_TC`—Count until terminal count and set the overflow status when it is reached. This mode is not available on the DAQ-STC.
- `COUNT_CONTINUOUSLY`—Count continuously. The Am9513 does not set the overflow status at terminal count, but the DAQ-STC does.

sourceEdge is the edge of the counter source or timebase signal on which it increments and accepts `COUNT_ON_RISING_EDGE` and `COUNT_ON_FALLING_EDGE` as values.

gateMode specifies how to use the signal on the counter GATE pin. You can use the following options:

- `UNGATED_SOFTWARE_START`—Ignore the gate signal and start when you call `CounterStart`.
- `COUNT_WHILE_GATE_HIGH`—Count while the gate signal is TTL high after you call `CounterStart`.
- `COUNT_WHILE_GATE_LOW`—Count while the gate signal is TTL low after you call `CounterStart`.

- `START_COUNTING_ON_RISING_EDGE`—Start counting on the rising edge of the TTL gate signal after you call `CounterStart`.
- `START_COUNTING_ON_FALLING_EDGE`—Start counting on the falling edge of the TTL gate signal after you call `CounterStart`.

CounterMeasureFrequency

```
short error = CounterMeasureFrequency (short device, char counter[],
                            unsigned short counterSize,
                            double gateWidthSampleTimeinSec,
                            double maxDelayBeforeGateSec,
                            unsigned short counterMinus1GateMode,
                            double *actualGateWidthSec, short *overflow,
                            short *valid, short *timeout, double *frequency);
```

Purpose

Measures the frequency of a TTL signal on the counter SOURCE pin you specify by counting rising edges of the signal during a period of time you specify. In addition to this connection, you also must wire the counter GATE pin to the OUT pin of **counter**–1. Table 10-6 defines **counter**, **counter**–1 and **counter**+1.

Table 10-6. Adjacent Counters

Chip	counter–1	counter	counter+1
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

`CounterMeasureFrequency` is useful for relatively high frequency signals when many cycles of the signal occur during the timing period. Use `PulseWidthOrPeriodMeasConfig` for relatively low-frequency signals. Remember that

period = 1/**frequency**

`CounterMeasureFrequency` configures **counter** and **counter**+1 (optional) to be event counters that count rising edges of the signal on the counter SOURCE pin. The function 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 reads the gate counter until its output state is low. `CounterMeasureFrequency` then reads the event counter and computes the signal frequency, number of events/actual gate pulse width, and stops the counters. You can also gate or trigger the operation with a signal on the **counter**–1 GATE pin.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
counter	string	Counter to use for the counting operation.
counterSize	unsigned short integer	Determines the size of the counter to use to perform the operation: `ONE_COUNTER` or `TWO_COUNTERS`.
gateWidthSampleTimeinSec	double	Desired length of the pulse to use to gate the signal. The lower the signal frequency, the longer the gate width must be.
maxDelayBeforeGateSec	double	Maximum expected delay between the time you call the function and the start of the gating pulse. If the gate signal does not start in this time, a timeout occurs.
counterMinus1GateMode	unsigned short integer	Gate mode for **counter**–1.

Output

Name	Type	Description
actualGateWidthSec	double	Achieved length in seconds of the gating pulse.
overflow	short integer	1 = counter rolled past terminal count 0 = counter did not roll past terminal count. If **overflow** is `1`, the value of **frequency** is inaccurate.
valid	short integer	Set to `1` if the measurement completes without a counter overflow. A timeout and a valid measurement can occur at the same time. A timeout does not produce an error.
timeout	short integer	Set to `1` if the time limit expires during the function call. A timeout and a valid measurement can occur at the same time. A timeout does not produce an error.
frequency	double	Frequency of the signal, it is computed as (number of rising edges) / (**actualGateWidthSec**).

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

counter is the counter to use for the counting operation. For valid counters, refer to Table 10-2 in the *Valid Counters for the Counter/Timer Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter.

counterSize determines the size of the counter to use to perform the operation. For a device with DAQ-STC counters, **counterSize** must be `ONE_COUNTER` (24-bit). For a device with Am9513 counters, **counterSize** can be `ONE_COUNTER` (16-bit) or `TWO_COUNTERS` (32-bit). If you use `TWO_COUNTERS`, **counter**+1 is cascaded with the specified counter. **counter**+1 is defined as shown in Table 10-6 in this function description.

counterMinus1GateMode is the gate mode for **counter**–1. The possible values are `UNGATED_SOFTWARE_START`, `COUNT_WHILE_GATE_HIGH`, `COUNT_WHILE_GATE_LOW`, and `START_COUNTING_ON_RISING_EDGE`

`CounterMeasureFrequency` uses **counter**–1 to gate **counter** so that rising edges are counted over a precise sample time. For a specified **counter**, **counter**–1 is defined as shown in Table 10-6 in this function description.

CounterRead

```
short error = CounterRead (unsigned long taskID, short *overflow,
                           long *count);
```

Purpose

Reads the counter **taskID** identifies.

Parameters

Input

Name	Type	Description
taskID	unsigned long integer	Reference number assigned to the counting operation by one of the counter configuration functions.

Output

Name	Type	Description
overflow	short integer	1 = counter rolled past terminal count 0 = counter did not roll past terminal count
count	long integer	Value of the counter at the time it is read.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

overflow indicates whether the counter rolled over past its terminal count. If **overflow** is `1`, the value of **count** is inaccurate.

CounterStart

```
short error = CounterStart (unsigned long taskID);
```

Purpose

Starts the counter **taskID** identifies.

Parameter

Input

Name	Type	Description
taskID	unsigned long integer	Reference number assigned to the counting operation by one of the counter configuration functions.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

CounterStop

```
short error = CounterStop (unsigned long taskID);
```

Purpose

Stops a count operation immediately.

Parameter

Input

Name	Type	Description
taskID	unsigned long integer	Reference number assigned to the counting operation by one of the counter configuration functions.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

DelayedPulseGenConfig

```
short error = DelayedPulseGenConfig (short device, char counter[],
                            double pulseDelay, double pulseWidth,
                            unsigned short timebaseSource,
                            unsigned short gateMode,
                            unsigned short pulsePolarity,
                            double *actualDelay, double *actualPulseWidth,
                            unsigned long *taskID);
```

Purpose

Configures a counter to generate a delayed TTL pulse or triggered pulse train on its OUT pin.

`DelayedPulseGenConfig` creates the signal by decrementing the counter twice, first for the delay to the pulse, phase 1, then for the pulse itself, phase 2. `DelayedPulseGenConfig` selects the highest resolution timebase to achieve the desired characteristics.

You can gate or trigger the operation with a signal on the counter GATE pin. Call `CounterStart` to start the operation or to enable the gate or trigger action.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
counter	string	Counter to use for the counting operation.
pulseDelay	double	Desired duration of the delay, phase 1, before the pulse.
pulseWidth	double	Desired duration of the pulse, phase 2, after the delay.
timebaseSource	unsigned short integer	Signal that causes the counter to count.
gateMode	unsigned short integer	Specifies how to use the signal on the counter GATE pin.
pulsePolarity	unsigned short integer	Polarity of phase 2 of each cycle.

Output

Name	Type	Description
actualDelay	double	Achieved delay based on the resolution and range of your hardware.
actualPulseWidth	double	Achieved pulse width based on the resolution and range of your hardware.
taskID	unsigned long integer	Reference number assigned to this operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

counter is the counter to use for the counting operation. For valid counters, refer to Table 10-2 in the *Valid Counters for the Counter/Timer Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter.

pulseDelay is the desired duration of the delay, phase 1, before the pulse. The unit is seconds if **timebaseSource** is `USE_INTERNAL_TIMEBASE` or cycles if **timebaseSource** is `USE_COUNTER_SOURCE`. If **pulseDelay** = `0.0` and **timebaseSource** is internal, `DelayedPulseGenConfig` selects a minimum delay of three cycles of the timebase you use.

pulseWidth is the desired duration of the pulse, phase 2, after the delay.

timebaseSource is the signal that causes the counter to count and can be one of the following values:

- `USE_INTERNAL_TIMEBASE`—`DelayedPulseGenConfig` selects an internal timebase based on the pulse delay and width, in units of seconds.
- `USE_COUNTER_SOURCE`—`DelayedPulseGenConfig` uses the signal on the counter SOURCE pin; the units of pulse delay and width are cycles of that signal.

gateMode specifies how to use the signal on the counter GATE pin. You can use the following options:

- `UNGATED_SOFTWARE_START`—Ignore the gate signal and start when you call `CounterStart`.
- `COUNT_WHILE_GATE_HIGH`—Count while the gate signal is TTL high after you call `CounterStart`.
- `COUNT_WHILE_GATE_LOW`—Count while the gate signal is TTL low after you call `CounterStart`.
- `START_COUNTING_ON_RISING_EDGE`—Start counting on the rising edge of the TTL gate signal after you call `CounterStart`.
- `START_COUNTING_ON_FALLING_EDGE`—Start counting on the falling edge of the TTL gate signal after you call `CounterStart`.
- `RESTART_ON_EACH_RISING_EDGE`—Restart counting on each rising edge of the TTL gate signal after you call `CounterStart`.
- `RESTART_ON_EACH_FALLING_EDGE`—Restart counting on each falling edge of the TTL gate signal after you call `CounterStart`.

pulsePolarity is the polarity of phase 2 of each cycle. You can use the following options:

- `POSITIVE_POLARITY`—The delay, phase 1, is a low TTL level, and the pulse, phase 2, is a high level.
- `NEGATIVE_POLARITY`—The delay, phase 1, is a high TTL level; and the pulse, phase 2, is a low level.

FrequencyDividerConfig

```
short error = FrequencyDividerConfig (short device, char counter[],
                          double sourceTimebase, double timebaseDivisor,
                          unsigned short gateMode,
                          unsigned short outputBehavior, short sourceEdge,
                          unsigned long *taskID);
```

Purpose

Configures the counter you specify to count the number of signal transitions on its SOURCE pin or on an internal timebase signal and to strobe or toggle the signal on its OUT pin.

To divide an external TTL signal, connect it to the counter SOURCE pin and set the **sourceTimebase** parameter to `USE_COUNTER_SOURCE`.

To divide an internal timebase signal, set the **sourceTimebase** parameter to a desired valid frequency.

Set the **timebaseDivisor** to the desired value. For a value of *n* and a pulsed output, an output pulse equal to the period of the source or timebase signal appears on the counter OUT pin once for each *n* cycles of that signal. For a toggled output, the output toggles after each *n* cycles. The toggled output frequency is half that of the pulsed output.

If **gateMode** is not `UNGATED_SOFTWARE_START`, connect your gate signal to the counter GATE pin.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
counter	string	Counter to use for the counting operation.
sourceTimebase	double	Use `USE_COUNTER_SOURCE` to count TTL edges at the counter SOURCE pin or supply a valid internal timebase frequency to count the TTL edges of an internal clock.
timebaseDivisor	double	Source frequency divisor.
gateMode	unsigned short integer	Specifies how to use the signal on the counter GATE pin.

Name	Type	Description
outputBehavior	unsigned short integer	Behavior of the output signal when counter reaches terminal count.
sourceEdge	short integer	Edge of the counter source or timebase signal on which it decrements: `COUNT_ON_RISING_EDGE` or `COUNT_ON_FALLING_EDGE`.

Output

Name	Type	Description
taskID	unsigned long integer	Reference number assigned to this operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

counter is the counter to use for the counting operation. For valid counters, refer to Table 10-2 in the *Valid Counters for the Counter/Timer Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter.

sourceTimebase determines whether the counter uses its SOURCE pin or an internal timebase as its signal source. Pass `USE_COUNTER_SOURCE` to count TTL edges at the counter SOURCE pin or pass a valid internal timebase frequency to count the TTL edges of an internal clock.

Table 10-7 shows valid internal timebase frequencies.

Table 10-7. Valid Internal Timebase Frequencies

Frequency	Chip Type on DAQ Board
1,000,000	Am9513
100,000	Am9513
10,000	Am9513
1,000	Am9513
100	Am9513
20,000,000	DAQ-STC
100,000	DAQ-STC

timebaseDivisor is the source frequency divisor. For example, if the source signal is 1,000 Hz, the **timebaseDivisor** is 10, and the output is pulsed, the frequency of the counter OUT signal is 100 Hz. If the output is toggled, the frequency is 50 Hz.

gateMode specifies how to use the signal on the counter GATE pin. You can use the following options:

- `UNGATED_SOFTWARE_START`—Ignore the gate signal and start when you call `CounterStart`.
- `COUNT_WHILE_GATE_HIGH`—Count while the gate signal is TTL high after you call `CounterStart`.
- `COUNT_WHILE_GATE_LOW`—Count while the gate signal is TTL low after you call `CounterStart`.
- `START_COUNTING_ON_RISING_EDGE`—Start counting on the rising edge of the TTL gate signal after you call `CounterStart`.
- `START_COUNTING_ON_FALLING_EDGE`—Start counting on the falling edge of the TTL gate signal after you call `CounterStart`.

outputBehavior is the behavior of the output signal when counter reaches terminal count and can be one of the following values:

- `HIGH_PULSE`—High pulse that lasts one cycle of the source or timebase signal.
- `LOW_PULSE`—Low pulse that lasts one cycle of the source or timebase signal.
- `HIGH_TOGGLE`—High toggle that lasts until the next terminal count (TC).
- `LOW_TOGGLE`—Low toggle that lasting until the next TC.

For a Timebase Divisor of N and a pulsed output, an output pulse equal to the period of the source or timebase signal appears on the counter OUT pin once each N cycles of that signal For a toggled output, the output toggles after each N cycles. The toggled output frequency is thus half that of the pulsed output, in other words,

$$\text{pulsedFrequency} = \text{sourceFrequency}/N$$

and

$$\text{toggledFrequency} = \text{sourceFrequency}/(2 \times N)$$

If N = 3, the OUT pin generates pulses as shown in Figure 10-4.

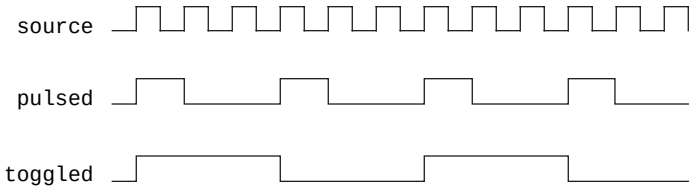

Figure 10-4. OUT Pin Pulses

GetAILimitsOfChannel

```
short error = GetAILimitsOfChannel (short device, char channelString[],
                                    char singleChannel[],
                                    double initialHighLimit,
                                    double initialLowLimit,
                                    double *highLimit, double *lowLimit);
```

Purpose

Returns the high and low limits for a particular channel in the channel string.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
channelString	string	Analog input channels to sample.
singleChannel	string	Single channel of the channel string.
initialHighLimit	double	Maximum value to measure for all channels that appear in the channel string before the first command string that specifies a new high limit.
initialLowLimit	double	Minimum value to measure for all channels that appear in the channel string before the first command string that specifies a new low limit.

Output

Name	Type	Description
highLimit	double	Returns the high limit for the channel you specify.
lowLimit	double	Returns the low limit for the channel you specify.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the list of Analog input channels to sample. Refer to the *Channel String for Analog Input Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

singleChannel is a single channel of the channel string. For example, if the channel string is `"0:3,5"` a single channel can be `"2"` or `"5"` and so on.

initialHighLimitVolts specifies the maximum voltage to measure for all channels that appear in the channel string before the first command string that specifies a new high limit. Consider the following channel string:

```
"0,1; cmd hi 10.0 low -10.0; 2,3"
```

If **initialHighLimitVolts** is 5.0, channels `"0"` and `"1"` have a high limit of `5.0` and channels `"2"` and `"3"` have a high limit of `10.0`.

initialLowLimitVolts is the minimum voltage to measure for all channels that appear in the channel string before the first command string that specifies a new low limit. Consider the following channel string:

```
"0,1; cmd hi 10.0 low -10.0; 2,3"
```

If the **initialLowLimitVolts** is `-5.0`, channels `"0"` and `"1"` have a low limit of `-5.0`, and channels `"2"` and `"3"` have a low limit of `-10.0`.

GetChannelIndices

```
short error = GetChannelIndices (short device, char channelString[],
                                 char channelSubString[], short channelType,
                                 long channelIndices[]);
```

Purpose

Determines the indices of the channels in **channelSubString**. For example, if **channelString** is `"1:6"` and **channelSubString** is `"1,3,6"`.

`GetChannelIndices` fills in the **channelIndices** array as follows:
channelIndices[0] = `0`;
channelIndices[1] = `2`;
channelIndices[2] = `5`;

`GetChannelIndices` is useful if you want to verify that a particular channel is part of **channelString**.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
channelString	string	Analog channel string.
channelSubString	string	Sub-string of **channelString**.
channelType	short integer	Specifies whether the **channelString** is `ANALOG_INPUT` or `ANALOG_OUTPUT`.

Output

Name	Type	Description
channelIndices	long integer array	Returns the indices of the channels in the **channelSubString**.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the analog channels to sample. Refer to the *Channel String for Analog Input Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

channelSubString is a sub-string of the **channelString**. For example, if **channelString** is `"0:3,5"` the sub-string can be `"2"` or `"1,3"`.

GetChannelNameFromIndex

```
short error = GetChannelNameFromIndex (short device, char channelString[],
                                long index, short channelType,
                                char channelName[]);
```

Purpose

Determines the name of the particular channel in **channelString** that **index** indicates.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by the NI-DAQ configuration utility.
channelString	string	Analog input channels to sample.
index	long integer	Index of a particular channel in **channelString**.
channelType	short integer	Specifies whether the **channelString** is `ANALOG_INPUT` or `ANALOG_OUTPUT`.

Output

Name	Type	Description
channelName	string	Returns the name of the particular channel in **channelString** that **index**.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the analog channels to sample. Refer to the *Channel String for Analog Input Functions* or *Channel String for Analog Output Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

channelName returns the name of the particular channel in **channelString** that **index** indicates. Declare this string to have `MAX_CHANNEL_NAME_LENGTH` bytes.

GetDAQErrorString

```
char *errorString = GetDAQErrorString (short errorNumber);
```

Purpose

Returns a string that contains the description for the numeric error code.

Parameter

Input

Name	Type	Description
errorNumber	short integer	Error number an Easy I/O for DAQ function returns.

Return Value

Name	Type	Description
errorString	string	String that contains the description for the numeric error code.

GetNumChannels

```
short error = GetNumChannels (short device, char channelString[],
                              short channelType,
                              unsigned long *numberOfChannels);
```

Purpose

Determines the number of channels **channelString** contains.

You must know the number of channels in **channelString** so you can correctly interpret analog input waveform arrays or build analog output waveform arrays.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
channelString	string	Analog channel string.
channelType	short integer	Specifies whether **channelString** is `ANALOG_INPUT` or `ANALOG_OUTPUT`.

Output

Name	Type	Description
numberOfChannels	unsigned long integer	Returns the number of channels **channelString** contains.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

channelString is the analog channels to sample. Refer to the *Channel String for Analog Input Functions* or *Channel String for Analog Output Functions* section of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

GroupByChannel

```
short error = GroupByChannel (float array[], long numberOfScans,
                              unsigned long numberOfChannels);
```

Purpose

Reorders an array of data from GROUPED_BY_SCAN mode into GROUPED_BY_CHANNEL mode.

If you acquire data in GROUPED_BY_SCAN mode, you need to reorder the array into “grouped by channel” mode before you can pass it to graph plotting functions, analysis functions, and others.

Refer to the description of the **fillMode** parameter of AIAcquireWaveforms for an explanation of GROUPED_BY_SCAN versus GROUPED_BY_CHANNEL.

Parameters

Input/Output

Name	Type	Description
array	double array	Pass in the GROUPED_BY_SCAN array. GroupByChannel groups the data by channel in place.

Input

Name	Type	Description
numberOfScans	long integer	Number of scans the data array contains.
numberOfChannels	unsigned long integer	Specifies the number of channels that were scanned. You can use GetNumChannels to determine the number of channels your channel string contains.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

ICounterControl

```
short error = ICounterControl (short device, short counter,
                               short controlCode, unsigned short count,
                               short binaryorBCD, short outputState,
                               unsigned short *readValue);
```

Purpose

Controls counters on devices that use the 8253 timer chip, such as Lab boards, SCXI-1200, DAQPad-1200, PC-LPM-16, and DAQCard 700.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
counter	short integer	Counter to control. Valid counters = 0–2.
controlCode	short integer	Determines the counter operating mode.
count	unsigned short integer	Period between output pulses.
binaryorBCD	short integer	`I_BINARY` = Counter operates as a 16-bit binary counter; 0–65,535 `I_BCD` = Counter operates as a 4-digit BCD counter; 0–9,999
outputState	short integer	`I_HIGH_STATE` = Output state of the counter is high `I_LOW_STATE` = Output state of the counter is low Matters only when **controlCode** = `7` (`I_RESET`).

Output

Name	Type	Description
readValue	unsigned short integer	Returns the value read from the counter when **controlCode** = `6` (`I_READ`).

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

controlCode determines the counter operating mode and accepts the following attributes:

- `0`: `I_TOGGLE_ON_TC`—Counter output becomes low after the mode set operation and the counter decrements from **count** to `0` while the gate is high. The output toggles from low to high after the counter reaches `0`.
- `1`: `I_PROGRAMMABLE_ONE_SHOT`—Counter output becomes low on the count following the leading edge of the gate input and becomes high on terminal count.
- `2`: `I_RATE_GENERATOR`—Counter output becomes low for one period of the clock input. **count** indicates the period between output pulses.
- `3`: `I_SQUARE_WAVE_RATE_GENERATOR`—Counter output stays high for half the **count** clock pulses and stays low for the other half.
- `4`: `I_SOFTWARE_TRIGGERED_STROBE`—Counter output is initially high, and the counter begins to count down while the gate input is high. On terminal count, the output becomes low for on clock pulse, then becomes high again.
- `5`: `I_HARDWARE_TRIGGERED_STROBE`—Similar to mode 4 except that a rising edge at the gate input triggers the count to start.
- `6`: `I_READ`—Reads the counter and returns the value in **readValue**.
- `7`: `I_RESET`—Resets the counter and sets its output to **outputState**.

count is the period between output pulses and can be one of the following values:

- If **controlCode** is 0, 1, 4, or 5, **count** can be 0 through 65,535 in binary counter operation and 0 through 9,999 in binary-coded decimal (BCD) counter operation.
- If **controlCode** is 2 or 3, **count** can be 2 through 65,535 in binary counter operation and 2 through 9,999 in BCD counter operation.

Note ***0 is equivalent to 65,536 in binary counter operation and 10,000 in BCD counter operation.***

PlotLastAIWaveformsPopup

```
short error = PlotLastAIWaveformsPopup (short device,
                          double waveformsBuffer[]);
```

Purpose

Plots the last analog input (AI) waveform you acquired. It is intended for demonstration purposes.

You must group data by channel before you pass it to `PlotLastAIWaveformsPopup`. Use `GROUP_BY_CHANNEL` as **fillMode** when you acquire the data, or call `GroupByChannel` before you call this function.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
waveformsBuffer	double array	Array that contains the last AI waveform acquired.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

PulseWidthOrPeriodMeasConfig

```
short error = PulseWidthOrPeriodMeasConfig (short device, char counter[],
                          unsigned short typeOfMeasurement,
                          double sourceTimebase, unsigned long *taskID);
```

Purpose

Configures the counter you specify to measure the pulse width or period of a TTL signal connected to its GATE pin. `PulseWidthOrPeriodMeasConfig` takes the measurement by counting the number of cycles of the timebase you specify between the appropriate starting and ending events.

Connect the signal you want to measure to the counter GATE pin.

To measure with an internal timebase, set **sourceTimebase** to the desired frequency.

To measure with an external timebase, connect that signal to the counter SOURCE pin and set **sourceTimebase** to `USE_COUNTER_SOURCE`.

Call `CounterStart` to start the measurement. Then call `CounterRead` to read the value. If the operation is valid, `CounterRead` returns a count greater than `3` in the **count** output parameter and returns `0` in the **overflow** output parameter.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
counter	string	Counter to use for the counting operation.
typeOfMeasurement	unsigned short integer	Identifies the type of pulse width or period measurement to make.
sourceTimebase	double	Use `USE_COUNTER_SOURCE` to count TTL edges at the counter SOURCE pin or supply a valid internal timebase frequency to count the TTL edges of an internal clock.

Output

Name	Type	Description
taskID	unsigned long integer	Reference number assigned to this counter operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

typeOfMeasurement identifies the type of pulse width or period measurement to make and accepts the following attributes:

- `MEASURE_HIGH_PULSE_WIDTH`—Measure high pulse width from rising to falling edge.
- `MEASURE_LOW_PULSE_WIDTH`—Measure low pulse width from falling to rising edge.
- `MEASURE_PERIOD_BTW_RISING_EDGES`—Measure period between adjacent rising edges.
- `MEASURE_PERIOD_BTW_FALLING_EDGES`—Measure period between adjacent falling edges.

sourceTimebase determines whether the counter uses its SOURCE pin or an internal timebase as its signal source. Pass `USE_COUNTER_SOURCE` to count TTL edges at the counter SOURCE pin or pass a valid internal timebase frequency to count the TTL edges of an internal clock.

Table 10-8 shows valid internal timebase frequencies and the corresponding chip types on a DAQ board.

Table 10-8. Valid Internal Timebase Frequencies

Frequency	Chip Type on DAQ Board
1,000,000	Am9513
100,000	Am9513
10,000	Am9513
1,000	Am9513

Table 10-8. Valid Internal Timebase Frequencies (Continued)

Frequency	Chip Type on DAQ Board
100	Am9513
20,000,000	DAQ-STC
100,000	DAQ-STC

ReadFromDigitalLine

```
short error = ReadFromDigitalLine (short device, char portNumber[],
                                   short line, short portWidth, long configure,
                                   unsigned long *lineState);
```

Purpose

Reads the logical state of a digital line on a port that you configure as input.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
portNumber	string	Specifies the digital port `ReadFromDigitalLine` configures.
line	short integer	Specifies the individual bit or line within the port to use for I/O (zero-based).
portWidth	short integer	Total width in bits of the port.
configure	long integer	`1` = Configure the digital port before reading `0` = Do not configure the digital port before reading

Output

Name	Type	Description
lineState	unsigned long integer	Returns the state of the digital line. `1` = logical high 0 = logical low

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

portNumber specifies the digital port `ReadFromDigitalLine` configures.

A **portNumber** value of `0` signifies port 0; a **portNumber** of `1` signifies port 1; and so on. If you use an SCXI-1160, SCXI-1161, SCXI-1162, or SCXI-1163 module, use the following syntax:

`"SCx!MDy!0"`

where *x* is the chassis ID and *y* is the module device number, to specify the port on a module.

portWidth is the total width in bits of the port. For example, you can combine two 4-bit ports into an 8-bit port on an MIO (non-E Series) board by setting **portWidth** to `8`.

When **portWidth** is greater than the physical port width of a digital port, the following restrictions apply: The **portWidth** must be an integral multiple of the physical port width, and the port numbers in the combined port must begin with the port named by **portNumber** and must increase consecutively. For example, if **portNumber** is 3 and **portWidth** is 24 (bits), LabWindows/CVI uses ports 3, 4, and 5.

You must pass a **portWidth** of at least 8 for the 8255-based digital I/O ports, including all digital ports on Lab boards, SCXI-1200, DAQPad-1200, DAQCard-1200, DIO-24, DIO-32F, DIO-96, and AT-MIO-16DE-10/AT-MIO-16D ports 2, 3, and 4.

configure specifies whether to configure the digital port before reading.

When you call `ReadFromDigitalLine` in a loop, you can optimize it by configuring the digital port only on the first iteration.

When you configure a digital I/O port that is part of an 8255 PPI, including all digital ports on Lab boards, SCXI-1200, DAQPad-1200, DAQCard-1200, DIO-24, DIO-32F, DIO-96, and AT-MIO-16DE-10/AT-MIO-16D ports 2, 3, and 4, the 8255 PPI goes through a configuration phase in which all the ports within the same PPI chip are reset to logic low, regardless of the data direction. The data directions on other ports, however, are maintained.

ReadFromDigitalPort

```
short error = ReadFromDigitalPort (short device, char portNumber[],
                                   short portWidth, long configure,
                                   unsigned long *pattern);
```

Purpose

Reads a digital port that you configure for input.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
portNumber	string	Specifies the digital port `ReadFromDigitalPort` configures.
portWidth	short integer	Total width in bits of the port.
configure	long integer	`1`= Configure the digital port before reading `0` = Do not configure the digital port before reading

Output

Name	Type	Description
pattern	unsigned long integer	Data read from the digital port.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

portNumber specifies the digital port `ReadFromDigitalPort` configures.

A **portNumber** value of `0` signifies port 0; a **portNumber** of `1` signifies port 1; and so on. If you use an SCXI-1160, SCXI-1161, SCXI-1162, or SCXI-1163 module, use the following syntax:

`"SCx!MDy!0"`

where *x* is the chassis ID and *y* is the module device number, to specify the port on a module.

portWidth is the total width in bits of the port. For example, you can combine two 4-bit ports into an 8-bit port on an MIO (non-E Series) board by setting **portWidth** to `8`.

When **portWidth** is greater than the physical port width of a digital port, the following restrictions apply: The **portWidth** must be an integral multiple of the physical port width, and the port numbers in the combined port must begin with the port named by **portNumber** and must increase consecutively. For example, if **portNumber** is 3 and **portWidth** is 24 (bits), LabWindows/CVI uses ports 3, 4, and 5.

You must pass a **portWidth** of at least 8 for the 8255-based digital I/O ports, including all digital ports on Lab boards, SCXI-1200, DAQPad-1200, DAQCard-1200, DIO-24, DIO-32F, DIO-96, and AT-MIO-16DE-10/AT-MIO-16D ports 2, 3, and 4.

configure specifies whether to configure the digital port before reading.

When you call `ReadFromDigitalPort` in a loop, you can optimize it by configuring the digital port only on the first iteration.

When you configure a digital I/O port that is part of an 8255 PPI, including all digital ports on Lab boards, SCXI-1200, DAQPad-1200, DAQCard-1200, DIO-24, DIO-32F, DIO-96, and AT-MIO-16DE-10/AT-MIO-16D ports 2, 3, and 4, the 8255 PPI goes through a configuration phase in which all the ports within the same PPI chip are reset to logic low, regardless of the data direction. The data directions on other ports, however, are maintained.

SetEasyIOMultitaskingMode

```
void SetEasyIOMultitaskingMode (int multitaskingMode);
```

Purpose

By default, if you call the non-timed Easy I/O for DAQ functions repetitively, LabWindows/CVI does not reconfigure the hardware unless you change the parameters. Thus, LabWindows/CVI improves the performance of these functions by reconfiguring the hardware only when necessary.

However, if you run multiple data acquisition programs simultaneously, the non-timed Easy I/O for DAQ functions do not know when the hardware has been reconfigured by another application that accesses the same DAQ device. Consequently, the functions might work incorrectly.

To get around this problem, you can force these functions to always reconfigure the hardware. You do this by setting the multitasking mode to `MULTITASKING_AWARE`.

You should set the multitasking mode to `MULTITASK_AWARE` if your program calls the non-timed Easy I/O for DAQ functions and you expect another data acquisition program to access the same board while your program runs. In this mode, the Easy I/O for DAQ functions always reconfigure the hardware on each invocation, which means they are not optimized for speed, but other applications cannot adversely affect them.

You should set the multitasking mode to `MULTITASK_UNAWARE` if you know no another program accesses the same DAQ device while your program runs.

Parameter

Input

Name	Type	Description
multitaskingMode	integer	When set to a nonzero value, DAQ devices are reconfigured to default settings every time an Easy I/O for DAQ function invokes such devices.

Return Value

None.

WriteToDigitalLine

```
short error = WriteToDigitalLine (short device, char portNumber[],
                                  short line, short portWidth, long configure,
                                  unsigned long lineState);
```

Purpose

Sets the output logic state of a digital line on a digital port.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
portNumber	string	Specifies the digital port this function configures.
line	short integer	Specifies the individual bit or line within the port to use for I/O.
portWidth	short integer	Total width in bits of the port.
configure	long integer	`1`= Configure the digital port before writing `0` = Do not configure the digital port before writing
lineState	unsigned long integer	Specifies the new state of the digital line. `1` = logical high `0` = logical low

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

portNumber specifies the digital port `WriteToDigitalLine` configures.

A **portNumber** value of `0` signifies port 0; a **portNumber** of `1` signifies port 1; and so on. If you use an SCXI-1160, SCXI-1161, SCXI-1162, or SCXI-1163 module, use the following syntax:

`"SCx!MDy!0"`

where *x* is the chassis ID and *y* is the module device number, to specify the port on a module.

portWidth is the total width in bits of the port. For example, you can combine two 4-bit ports into an 8-bit port on an MIO (non-E Series) board by setting **portWidth** to `8`.

When **portWidth** is greater than the physical port width of a digital port, the following restrictions apply: the **portWidth** must be an integral multiple of the physical port width, and the port numbers in the combined port must begin with the port named by **portNumber** and must increase consecutively. For example, if **portNumber** is 3 and **portWidth** is 24 (bits), LabWindows/CVI uses ports 3, 4, and 5.

You must pass a **portWidth** of at least 8 for the 8255-based digital I/O ports, including all digital ports on Lab boards, SCXI-1200, DAQPad-1200, DAQCard-1200, DIO-24, DIO-32F, DIO-96, and AT-MIO-16DE-10/AT-MIO-16D ports 2, 3, and 4.

configure specifies whether to configure the digital port before writing.

When you call `WriteToDigitalLine` in a loop, you can optimized it by configuring the digital port only on the first iteration.

When you configure a digital I/O port that is part of an 8255 PPI, including all digital ports on Lab boards, SCXI-1200, DAQPad-1200, DAQCard-1200, DIO-24, DIO-32F, DIO-96, and AT-MIO-16DE-10/AT-MIO-16D ports 2, 3, and 4, the 8255 PPI goes through a configuration phase in which all the ports within the same PPI chip are reset to logic low, regardless of the data direction. The data directions on other ports, however, are maintained.

WriteToDigitalPort

```
short error = WriteToDigitalPort (short device, char portNumber[],
                                  short portWidth, long configure,
                                  unsigned long pattern);
```

Purpose

Outputs a decimal pattern to a digital port.

Parameters

Input

Name	Type	Description
device	short integer	Assigned by NI-DAQ configuration utility.
portNumber	string	Specifies the digital port this function configures.
portWidth	short integer	Total width in bits of the port.
configure	long integer	`1` = Configure the digital port before writing `0` = Do not configure the digital port before writing
pattern	unsigned long integer	Specifies the new state of the lines in the port.

Return Value

Name	Type	Description
error	short integer	Refer to Table 10-9 for error codes.

Parameter Discussion

portNumber specifies the digital port `WriteToDigitalPort` configures.

A **portNumber** value of `0` signifies port 0; a **portNumber** of `1` signifies port 1; and so on. If you use an SCXI-1160, SCXI-1161, SCXI-1162, or SCXI-1163 module, use the following syntax:

`"SCx!MDy!0"`

where *x* is the chassis ID and *y* is the module device number, to specify the port on a module.

portWidth is the total width in bits of the port. For example, you can combine two 4-bit ports into an 8-bit port on an MIO (non-E Series) board by setting **portWidth** to `8`.

When **portWidth** is greater than the physical port width of a digital port, the following restrictions apply: the **portWidth** must be an integral multiple of the physical port width, and the port numbers in the combined port must begin with the port named by **portNumber** and must increase consecutively. For example, if **portNumber** is 3 and **portWidth** is 24 (bits), LabWindows/CVI uses ports 3, 4, and 5.

You must pass a **portWidth** of at least 8 for the 8255-based digital I/O ports, including all digital ports on Lab boards, SCXI-1200, DAQPad-1200, DAQCard-1200, DIO-24, DIO-32F, DIO-96, and AT-MIO-16DE-10/AT-MIO-16D ports 2, 3, and 4.

configure specifies whether to configure the digital port before writing.

When you call `WriteToDigitalPort` in a loop, you can optimized it by configuring the digital port only on the first iteration.

When you configure a digital I/O port that is part of an 8255 PPI, including all digital ports on Lab boards, SCXI-1200, DAQPad-1200, DAQCard-1200, DIO-24, DIO-32F, DIO-96, and AT-MIO-16DE-10/AT-MIO-16D ports 2, 3, and 4, the 8255 PPI goes through a configuration phase in which all the ports within the same PPI chip are reset to logic low, regardless of the data direction. The data directions on other ports, however, are maintained.

Error Conditions

All the functions in the Easy I/O for DAQ Library return an error code. A negative number indicates that an error occurred. If the return value is positive, it has the same description as if it were negative, but it is considered a warning.

Table 10-9. Easy I/O for DAQ Library Error Codes

Code	Constant Name	Explanation
0		Success.
–10001	**syntaxErr**	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	**semanticsErr**	An error was detected in the input string; the syntax of the string is correct, but certain values you specify in the string are inconsistent with other values you specify in the string.
–10003	**invalidValueErr**	The value of a numeric parameter is invalid.
–10004	**valueConflictErr**	The value of a numeric parameter is inconsistent with another parameter, and the combination is therefore invalid.
–10005	**badDeviceErr**	Device parameter is invalid.
–10006	**badLineErr**	Line parameter is invalid.
–10007	**badChanErr**	A channel is out of range for the device type or input configuration, the combination of channels is invalid, or you must reverse the scan order so that channel 0 is last.
–10008	**badGroupErr**	Group parameter is invalid.
–10009	**badCounterErr**	Counter parameter is invalid.
–10010	**badCountErr**	Count parameter is too small or too large for the specified counter.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10011	**badIntervalErr**	Interval parameter is too small or too large for the associated counter or I/O channel.
-10012	**badRangeErr**	Analog input or analog output voltage range is invalid for the specified channel.
-10013	**badErrorCodeErr**	Driver returned an unrecognized or unlisted error code.
-10014	**groupTooLargeErr**	Group size is too large for the device.
-10015	**badTimeLimitErr**	Time limit parameter is invalid.
-10016	**badReadCountErr**	Read count parameter is invalid.
-10017	**badReadModeErr**	Read mode parameter is invalid.
-10018	**badReadOffsetErr**	Offset is unreachable.
-10019	**badClkFrequencyErr**	Frequency parameter is invalid.
-10020	**badTimebaseErr**	Timebase parameter is invalid.
-10021	**badLimitsErr**	Limits are beyond the range of the device.
-10022	**badWriteCountErr**	Data array contains an incomplete update; 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 half the internal buffer size.
-10023	**badWriteModeErr**	Write mode is out of range or is invalid.
-10024	**badWriteOffsetErr**	Write offset plus the write mark is greater than the internal buffer size or it must be set to `0`.
-10025	**limitsOutOfRangeErr**	Voltage limits are out of range for this device in the current configuration. Alternate limits were selected.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10026	**badInputBufferSpecification**	Input buffer specification is invalid. This error results if, for example, you try to configure a multiple-buffer acquisition for a device that cannot perform multiple-buffer acquisition.
-10027	**badDAQEventErr**	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, with no remainder. If the TIO-10 is used for DAQEvent 4, general value `A` must be `1` or `2`.
-10028	**badFilterCutoffErr**	Cutoff frequency is not valid for this device.
-10080	**badGainErr**	Gain parameter is invalid.
-10081	**badPretrigCountErr**	Pretrigger sample count is invalid.
-10082	**badPosttrigCountErr**	Posttrigger sample count is invalid.
-10083	**badTrigModeErr**	Trigger mode is invalid.
-10084	**badTrigCountErr**	Trigger count is invalid.
-10085	**badTrigRangeErr**	Trigger range or trigger hysteresis window is invalid.
-10086	**badExtRefErr**	External reference value is invalid.
-10087	**badTrigTypeErr**	Trigger type parameter is invalid.
-10088	**badTrigLevelErr**	Trigger level parameter is invalid.
-10089	**badTotalCountErr**	Total count you specified is inconsistent with the buffer configuration and pretrigger scan count or with the device type.
-10090	**badRPGErr**	Individual range, polarity, and gain settings are valid but the combination you specified is invalid for this device.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10091	**badIterationsErr**	Analog output buffer iterations count is invalid. It must be 0, for indefinite iterations, or 1.
-10100	**badPortWidthErr**	Requested digital port width is not a multiple of the hardware port width.
-10240	**noDriverErr**	Driver interface could not locate or open the driver.
-10241	**oldDriverErr**	Driver is out of date.
-10242	**functionNotFoundErr**	Specified function is not located in the driver.
-10243	**configFileErr**	Driver could not locate or open the configuration file, or the format of the configuration file is not compatible with the currently installed driver.
-10244	**deviceInitErr**	Driver encountered a hardware-initialization error while attempting to configure the specified device.
-10245	**osInitErr**	Driver encountered an operating system error while attempting to perform an operation, or the driver performed an operation that the operating system does not recognize.
-10246	**communicationsErr**	Driver is unable to communicate with the specified external device.
-10247	**cmosConfigErr**	CMOS configuration memory for the computer is empty or invalid, or the configuration you specified does not agree with the current configuration of the computer.
-10248	**dupAddressErr**	Base addresses for two or more devices are the same; consequently, the driver is unable to access the specified device.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
`-10249`	**intConfigErr**	Interrupt configuration is incorrect given the capabilities of the computer or device.
`-10250`	**dupIntErr**	Interrupt levels for two or more devices are the same.
`-10251`	**dmaConfigErr**	DMA configuration is incorrect given the capabilities of the computer/DMA controller or device.
`-10252`	**dupDMAErr**	DMA channels for two or more devices are the same.
`-10253`	**switchlessBoardErr**	NI-DAQ was unable to find one or more switchless boards you configured using WDAQCONF.
`-10254`	**DAQCardConfigErr**	Cannot configure the DAQCard because: the correct version of card and socket services software is not installed; the card in the PCMCIA socket is not a DAQCard: or the base address and/or interrupt level you requested are not available according to the card and socket services resource manager. Try different settings or use `AutoAssign` in the NI-DAQ configuration utility.
`-10340`	**noConnectErr**	No RTSI signal/line is connected, or the specified signal and the specified line are not connected.
`-10341`	**badConnectErr**	RTSI signal/line cannot be connected as specified.
`-10342`	**multConnectErr**	Specified RTSI signal is already being driven by a RTSI line, or the specified RTSI line is already being driven by a RTSI signal.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
–10343	**SCXIConfigErr**	Specified SCXI configuration parameters are invalid, or the function cannot be executed given the current SCXI configuration.
–10360	**DSPInitErr**	DSP driver was unable to load the kernel for its operating system.
–10370	**badScanListErr**	Scan list is invalid. This error can result if, for example, you mix AMUX-64T channels and onboard channels, or if you scan multiplexed SCXI channels out of order.
–10400	**userOwnedRsrcErr**	Specified resource is owned by the user and cannot be accessed or modified by the driver.
–10401	**unknownDeviceErr**	Specified device is not a National Instruments product, or the driver does not work with the device. For example, the driver was released before the features of the device existed.
–10402	**deviceNotFoundErr**	No device is located in the specified slot or at the specified address.
–10403	**deviceSupportErr**	Requested action does not work with specified device. The driver recognizes the device, but the action is inappropriate for the device.
–10404	**noLineAvailErr**	No line is available.
–10405	**noChanAvailErr**	No channel is available.
–10406	**noGroupAvailErr**	No group is available.
–10407	**lineBusyErr**	Specified line is in use.
–10408	**chanBusyErr**	Specified channel is in use.
–10409	**groupBusyErr**	Specified group is in use.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10410	**relatedLCGBusyErr**	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	**counterBusyErr**	Specified counter is in use.
-10412	**noGroupAssignErr**	No group is assigned, or the specified line or channel cannot be assigned to a group.
-10413	**groupAssignErr**	A group is already assigned, or the specified line or channel is already assigned to a group.
-10414	**reservedPinErr**	Selected signal indicates a pin reserved by NI-DAQ. You cannot configure this pin yourself.
-10440	**sysOwnedRsrcErr**	Specified resource is owned by the driver and cannot be accessed or modified by the user.
-10441	**memConfigErr**	No memory is configured to work with the current data transfer mode, or the configured memory does not work with the current data transfer mode. If block transfers are in use, the memory must be capable of performing block transfers.
-10442	**memDisabledErr**	Specified memory is disabled or is unavailable given the current addressing mode.
-10443	**memAlignmentErr**	Transfer buffer is not aligned properly for the current data transfer mode. For example, the memory 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.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
–10444	**memFullErr**	No more system memory is available on the heap, or no more memory is available on the device.
–10445	**memLockErr**	Transfer buffer cannot be locked into physical memory.
–10446	**memPageErr**	Transfer buffer contains a page break. System resources might require reprogramming when the page break is encountered.
–10447	**memPageLockErr**	Operating environment is unable to grant a page lock.
–10448	**stackMemErr**	Driver is unable to continue parsing a string input because of stack limitations.
–10449	**cacheMemErr**	A cache-related error occurred, or caching does not work in the current mode.
–10450	**physicalMemErr**	A hardware error occurred in physical memory, or no memory is located at the specified address.
–10451	**virtualMemErr**	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	**noIntAvailErr**	No interrupt level is available for use.
–10453	**intInUseErr**	Specified interrupt level is already in use by another device.
–10454	**noDMACErr**	No DMA controller is available in the system.
–10455	**noDMAAvailErr**	No DMA channel is available for use.
–10456	**DMAInUseErr**	Specified DMA channel is already in use by another device.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10457	**badDMAGroupErr**	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.
-10459	**DLLInterfaceErr**	DLL could not be called because of an interface error.
-10460	**interfaceInteractionErr**	You have attempted to mix LabVIEW 2.2 VIs and LabVIEW 3.0 VIs. You must run an application that consists only of 2.2 VIs, then run the 2.2 Board Reset VI, before you can run any 3.0 VIs. You must run an application that consists of only 3.0 VIs, then run the 3.0 Device Reset VI, before you can run any 2.2 VIs.
-10560	**invalidDSPhandleErr**	DSP handle input to the VI is not a valid handle.
-10600	**noSetupErr**	No setup operation has been performed for the specified resources.
-10601	**multSetupErr**	Specified resources have already been configured by a setup operation.
-10602	**noWriteErr**	No output data has been written into the transfer buffer.
-10603	**groupWriteErr**	Output data associated with a group must be for a single channel or must be for consecutive channels.
-10604	**activeWriteErr**	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, all output channels currently using the DMA channel will require new data.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
–10605	**endWriteErr**	No data was written to the transfer buffer because the final data block has already been loaded.
–10606	**notArmedErr**	Specified resource is not armed.
–10607	**armedErr**	Specified resource is already armed.
–10608	**noTransferInProgErr**	No transfer is in progress for the specified resource.
–10609	**transferInProgErr**	A transfer is already in progress for the specified resource.
–10610	**transferPauseErr**	A single output channel in a group cannot be paused if the output data for the group is interleaved.
–10611	**badDirOnSomeLinesErr**	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	**badLineDirErr**	Specified line does not support the specified transfer direction.
–10613	**badChanDirErr**	Specified channel does not support the specified transfer direction.
–10614	**badGroupDirErr**	Specified group does not support the specified transfer direction.
–10615	**masterClkErr**	Clock configuration for the clock master is invalid.
–10616	**slaveClkErr**	Clock configuration for the clock slave is invalid.
–10617	**noClkSrcErr**	No source signal has been assigned to the clock resource.
–10618	**badClkSrcErr**	Specified source signal cannot be assigned to the clock resource.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10619	**multClkSrcErr**	A source signal has already been assigned to the clock resource.
-10620	**noTrigErr**	No trigger signal has been assigned to the trigger resource.
-10621	**badTrigErr**	Specified trigger signal cannot be assigned to the trigger resource.
-10622	**preTrigErr**	Pretrigger mode is not supported or is not available in the current configuration, or no pretrigger source has been assigned.
-10623	**postTrigErr**	No posttrigger source has been assigned.
-10624	**delayTrigErr**	Delayed trigger mode is not supported or is not available in the current configuration, or no delay source has been assigned.
-10625	**masterTrigErr**	Trigger configuration for the trigger master is invalid.
-10626	**slaveTrigErr**	Trigger configuration for the trigger slave is invalid.
-10627	**noTrigDrvErr**	No signal has been assigned to the trigger resource.
-10628	**multTrigDrvErr**	A signal has already been assigned to the trigger resource.
-10629	**invalidOpModeErr**	Specified operating mode is invalid, or the resources have not been configured for the specified operating mode.
-10630	**invalidReadErr**	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	**noInfiniteModeErr**	Continuous input or output transfers are invalid in the current operating mode.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10632	**someInputsIgnoredErr**	Certain inputs were ignored because they are not relevant in the current operating mode.
-10633	**invalidRegenModeErr**	This device does not support the specified analog output regeneration mode.
-10680	**badChanGainErr**	All channels must have an identical setting for this device.
-10681	**badChanRangeErr**	All channels of this device must have the same range.
-10682	**badChanPolarityErr**	All channels of this device must have the same polarity.
-10683	**badChanCouplingErr**	All channels of this device must have the same coupling.
-10684	**badChanInputModeErr**	All channels of this device must have the same input range.
-10685	**clkExceedsBrdsMaxConvRate**	Clock rate selected exceeds the recommended maximum rate for this device.
-10686	**scanListInvalidErr**	A configuration change has invalidated the scan list.
-10687	**bufferInvalidErr**	A configuration change has invalidated the allocated buffer.
-10688	**noTrigEnabledErr**	Total number of scans and pretrigger scans implies that a trigger start is intended, but no trigger is enabled.
-10689	**digitalTrigBErr**	Digital trigger B is illegal for the total scans and pretrigger scans specified.
-10690	**digitalTrigAandBErr**	With this device, you cannot enable digital triggers A and B at the same time.
-10691	**extConvRestrictionErr**	With this device, you cannot use an external sample clock with an external scan clock, start trigger, or stop trigger.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10692	**chanClockDisabledErr**	Cannot start the acquisition because the channel clock is disabled.
-10693	**extScanClockErr**	Cannot use an external scan clock when performing a single scan of a single channel.
-10694	**unsafeSamplingFreqErr**	Sampling frequency exceeds the safe maximum rate for the ADC, gains, and filters you are using.
-10695	**DMAnotAllowedErr**	You must use interrupts. DMA does not work.
-10696	**multiRateModeErr**	Multirate scanning can not be used with AMUX-64, SCXI, or pre-triggered acquisitions.
-10697	**rateNotSupportedErr**	NI-DAQ was unable to convert your timebase/interval pair to match the actual hardware capabilities of the specified board.
-10698	**timebaseConflictErr**	You cannot use this combination of scan and sample clock timebases for the specified board.
-10699	**polarityConflictErr**	You cannot use this combination of scan and sample clock source polarities for this operation for the specified board.
-10700	**signalConflictErr**	You cannot use this combination of scan and convert clock signal sources for this operation for the specified board.
-10740	**SCXITrackHoldErr**	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.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
-10780	**sc2040InputModeErr**	When you have an SC2040 attached to your device, all analog input channels must be configured for differential input mode.
-10800	**timeOutErr**	Operation could not complete within the time limit.
-10801	**calibrationErr**	An error occurred during the calibration process.
-10802	**dataNotAvailErr**	Requested amount of data has not yet been acquired, or the acquisition has completed and no more data is available to read.
-10803	**transferStoppedErr**	Transfer has been stopped to prevent regeneration of output data.
-10804	**earlyStopErr**	Transfer stopped before reaching the end of the transfer buffer.
-10805	**overRunErr**	Clock source for the input transfer is faster than the maximum input-clock rate; the integrity of the data has been compromised. Alternatively, the clock source for the output transfer is faster than the maximum output-clock rate; a data point was generated more than once because the update occurred before new data was available.
-10806	**noTrigFoundErr**	No trigger value was found in the input transfer buffer.
-10807	**earlyTrigErr**	Trigger occurred before sufficient pretrigger data was acquired.
-10809	**gateSignalErr**	Attempted to start a pulse width measurement with the pulse in the active state.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
–10840	**softwareErr**	Contents or the location of the driver file was changed between accesses to the driver.
–10841	**firmwareErr**	Firmware does not support the specified operation, or the firmware operation could not complete because of a data-integrity problem.
–10842	**hardwareErr**	Hardware is not responding to the specified operation, or the response from the hardware is not consistent with the functionality of the hardware.
–10843	**underFlowErr**	The update rate exceeds your system capacity to supply data to the output channel.
–10844	**underWriteErr**	At the time of the update for the device-resident memory, insufficient data was present in the output transfer buffer to complete the update.
–10845	**overFlowErr**	At the time of the update clock for the input channel, the device-resident memory was unable to accept additional data. One or more data points might have been lost.
–10846	**overWriteErr**	New data was written into the input transfer buffer before the old data was retrieved.
–10847	**dmaChainingErr**	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	**noDMACountAvailErr**	Driver could not obtain a valid reading from the transfer-count register in the DMA controller.
–10849	**openFileErr**	Unable to open a file.

Table 10-9. Easy I/O for DAQ Library Error Codes (Continued)

Code	Constant Name	Explanation
–10850	**closeFileErr**	Unable to close a file.
–10851	**fileSeekErr**	Unable to seek within a file.
–10852	**readFileErr**	Unable to read from a file.
–10853	**writeFileErr**	Unable to write to a file.
–10854	**miscFileErr**	An error occurred accessing a file.
–10880	**updateRateChangeErr**	A change to the update rate is not possible at this time because when waveform generation is in progress, you cannot change the interval timebase; or when you make several changes in a row, you must wait long enough for each change to take effect before you request more changes.
–10920	**gpctrDataLossErr**	One or more data points might have been lost during buffered GPCTR operations because of speed limitations of your system.

11

ActiveX Automation Library

This chapter describes the ActiveX Automation Library, which contains functions that control ActiveX Automation servers. The *ActiveX Automation Library Function Overview* section contains general information about the functions as well as guidelines and restrictions you should know when you use the ActiveX Automation Library. The *ActiveX Automation Library Function Reference* section contains an alphabetical list of function descriptions.

Note ***This library is available only on Windows 95/NT.***

ActiveX Automation Library Function Overview

ActiveX Automation (formerly called OLE Automation) allows applications to make their unique features available to scripting tools and other applications. An ActiveX Automation server is the application that exposes its features. An ActiveX Automation controller is the application that uses the features of an ActiveX Automation server. An ActiveX Automation server exports its features as a set of ActiveX Objects. For example, Microsoft Excel exposes its workbooks, worksheets, and charts as ActiveX Objects. Each ActiveX Object provides methods and properties that other applications can access. For example, the Microsoft Excel worksheet object provides a calculate method that calculates the values in a worksheet.

The ActiveX Automation Library contains functions that facilitate calling into ActiveX server interfaces. Use the ActiveX Automation Library in conjunction with the instrument drivers the ActiveX Automation Controller Wizard generates. The ActiveX Automation Controller Instrument Drivers contain C functions to create ActiveX Objects, to call ActiveX Object methods, and to get and set ActiveX Object properties. Select **Tools»Create ActiveX Automation Controller** to start the wizard.

The ActiveX Automation Library contains functions that:

- Help you work with the `VARIANT` parameters, `SAFEARRAY` parameters, and return values of the functions in the generated instrument drivers.
- Free resources dynamically allocated by the generated instrument drivers or by other ActiveX Automation Library functions.
- Display error information the library functions or the Automation server methods return.

The ActiveX Automation Library also contains low-level functions that the generated drivers use. These low-level functions invoke methods of server objects and get and set properties of server objects.

If you want to use the low-level functions, you should know ActiveX concepts. In particular, you should know how to browse through an ActiveX Automation server type library.

Variants and Safe Arrays

The `VARIANT` data type is a structure that can hold a value of any valid ActiveX Automation data type. Refer to Tables 11-1 and 11-2 for valid ActiveX Automation data types. ActiveX Automation server functions declare a parameter as a `VARIANT` when the parameter can take a value of more than one data type. This document uses the term *variant* to refer to parameters or variables declared with the `VARIANT` data type. The ActiveX Automation Library provides functions to help you pass values as variant input parameters and retrieve values from variant output parameters.

The `SAFEARRAY` data type is a structure that holds an array of data, the number of dimensions in the array, and the size of each dimension. ActiveX Automation server functions use the `SAFEARRAY` data type to pass arrays. This document uses the term *safe array* to refer to parameters or variables declared with the `SAFEARRAY` data type. The ActiveX Automation Library provides functions to convert between C-style arrays and safe arrays, functions to obtain the number of dimensions in a safe array and the size of each dimension, and functions to convert between C-arrays and safe arrays stored inside variants.

You can declare a variant structure as a local or global variable, but safe arrays are always dynamically allocated. Consequently, you must always reference safe arrays using pointers. Use the `LPSAFEARRAY` typedef to declare a pointer to a safe array. Microsoft adds `LP` at the beginning of data type names to indicate a pointer to a data type. Thus, `LPSAFEARRAY` signifies a pointer to a `SAFEARRAY`.

Events are Not Supported

The ActiveX Automation Library does not receive events from Automation servers. If you need to receive events through an Automation server event interface, you must manually create an IDispatch interface that conforms to the event interface the server provides.

ActiveX Automation Library Function Panels

The ActiveX Automation Library function panels are grouped in the tree structure in Table 11-1 according to the types of operations they perform.

The first- and second-level headings in the tree are names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings are the names of individual function panels. Each function panel generates a function call.

The following shows the structure of the ActiveX Automation Library function tree.

Variant-Related Functions
- Passing Values as Variants
- Assigning Values to Variants
- Querying the Type of a Variant
- Retrieving Values from Variants

Array Functions
- C Array-to-Safe Array Conversion
- Safe Array-to-C Array Conversion
- Querying Safe Arrays

BSTR Functions
Freeing Resources
Error Processing
Locales
Low-level Functions
- Creating Automation Objects
- Calling Methods and Properties

Table 11-1. Functions in the ActiveX Automation Library Function Tree

Class/Panel Name	Function Name
Variant-Related Functions	
Passing Values as Variants	
Variant From Long	`CA_VariantLong`
Variant From Short	`CA_VariantShort`
Variant From Int	`CA_VariantInt`
Variant From Bool	`CA_VariantBool`
Variant From Float	`CA_VariantFloat`
Variant From Double	`CA_VariantDouble`
Variant From Currency	`CA_VariantCurrency`
Variant From Date	`CA_VariantDate`
Variant From Error	`CA_VariantError`
Variant From UnsignedChar	`CA_VariantUChar`
Variant From Dispatch	`CA_VariantDispatch`
Variant From IUnknown	`CA_VariantIUnknown`
Variant From BSTR	`CA_VariantBSTR`
Empty Variant	`CA_VariantEmpty`
NULL Variant	`CA_VariantNULL`
Default Value Variant	`CA_DefaultValueVariant`
Assigning Values to Variants	
Variant Set Empty	`CA_VariantSetEmpty`
Variant Set Long	`CA_VariantSetLong`
Variant Set Short	`CA_VariantSetShort`
Variant Set Int	`CA_VariantSetInt`
Variant Set Bool	`CA_VariantSetBool`
Variant Set Float	`CA_VariantSetFloat`
Variant Set Double	`CA_VariantSetDouble`

Table 11-1. Functions in the ActiveX Automation Library Function Tree (Continued)

Class/Panel Name	Function Name
Variant-Related Functions (continued)	
Assigning Values to Variants (continued)	
Variant Set Safe Array	`CA_VariantSetSafeArray`
Variant Set 1D Array	`CA_VariantSet1DArray`
Variant Set 2D Array	`CA_VariantSet2DArray`
Variant Set BSTR	`CA_VariantSetBSTR`
Variant Set NULL	`CA_VariantSetNULL`
Variant Set Currency	`CA_VariantSetCurrency`
Variant Set Date	`CA_VariantSetDate`
Variant Set Dispatch	`CA_VariantSetDispatch`
Variant Set IUnknown	`CA_VariantSetIUnknown`
Variant Set Error	`CA_VariantSetError`
Variant Set Unsigned Char	`CA_VariantSetUChar`
Variant Set C String	`CA_VariantSetCString`
Variant Set Long Ptr	`CA_VariantSetLongPtr`
Variant Set Short Ptr	`CA_VariantSetShortPtr`
Variant Set Int Ptr	`CA_VariantSetIntPtr`
Variant Set Bool Ptr	`CA_VariantSetBoolPtr`
Variant Set Float Ptr	`CA_VariantSetFloatPtr`
Variant Set Double Ptr	`CA_VariantSetDoublePtr`
Variant Set Safe Array Ptr	`CA_VariantSetSafeArrayPtr`
Variant Set BSTR Ptr	`CA_VariantSetBSTRPtr`
Variant Set Currency Ptr	`CA_VariantSetCurrencyPtr`
Variant Set Date Ptr	`CA_VariantSetDatePtr`
Variant Set Dispatch Ptr	`CA_VariantSetDispatchPtr`
Variant Set IUnknown Ptr	`CA_VariantSetIUnknownPtr`
Variant Set Error Ptr	`CA_VariantSetErrorPtr`
Variant Set Unsigned Char Ptr	`CA_VariantSetUCharPtr`
Variant Set Variant Ptr	`CA_VariantSetVariantPtr`
Querying the Type of a Variant	
Variant Get Type	`CA_VariantGetType`
Variant Has Array	`CA_VariantHasArray`
Variant Has Pointer	`CA_VariantHasPtr`
Variant Has Long	`CA_VariantHasLong`
Variant Has Short	`CA_VariantHasShort`
Variant Has Int	`CA_VariantHasInt`
Variant Has Bool	`CA_VariantHasBool`
Variant Has Float	`CA_VariantHasFloat`
Variant Has Double	`CA_VariantHasDouble`
Variant Has C String	`CA_VariantHasCString`
Variant Has BSTR	`CA_VariantHasBSTR`
Variant Has NULL	`CA_VariantHasNull`
Variant Has Currency	`CA_VariantHasCurrency`
Variant Has Date	`CA_VariantHasDate`
Variant Has IUnknown	`CA_VariantHasIUnknown`

Table 11-1. Functions in the ActiveX Automation Library Function Tree (Continued)

Class/Panel Name	Function Name
Variant-Related Functions (continued)	
Querying the Type of a Variant (continued)	
Variant Has Dispatch	CA_VariantHasDispatch
Variant Has ObjHandle	CA_VariantHasObjHandle
Variant Has Unsigned Char	CA_VariantHasUChar
Variant Has Error Code	CA_VariantHasError
Variant Is Empty	CA_VariantIsEmpty
Retrieving Values from Variants	
Convert Variant To Type	CA_VariantConvertToType
Copy Variant	CA_VariantCopy
Variant Get Long	CA_VariantGetLong
Variant Get Short	CA_VariantGetShort
Variant Get Int	CA_VariantGetInt
Variant Get Boolean	CA_VariantGetBool
Variant Get Float	CA_VariantGetFloat
Variant Get Double	CA_VariantGetDouble
Variant Get Safe Array	CA_VariantGetSafeArray
Variant Get 1D Array	CA_VariantGet1DArray
Variant Get 2D Array	CA_VariantGet2DArray
Variant Get 1D Array in Buffer	CA_VariantGet1DArrayBuf
Variant Get 2D Array in Buffer	CA_VariantGet2DArrayBuf
Variant Get Array Num Dims	CA_VariantGetArrayNumDims
Variant Get 1D Array Size	CA_VariantGet1DArraySize
Variant Get 2D Array Size	CA_VariantGet2DArraySize
Variant Get BSTR	CA_VariantGetBSTR
Variant Get Currency	CA_VariantGetCurrency
Variant Get Date	CA_VariantGetDate
Variant Get Dispatch	CA_VariantGetDispatch
Variant Get IUnknown	CA_VariantGetIUnknown
Variant Get Error	CA_VariantGetError
Variant Get Unsigned Char	CA_VariantGetUChar
Variant Get ObjHandle	CA_VariantGetObjHandle
Variant Get String Length	CA_VariantGetCStringLen
Variant Get String In Buffer	CA_VariantGetCStringBuf
Variant Get String	CA_VariantGetCString
Variant Get Long Ptr	CA_VariantGetLongPtr
Variant Get Short Ptr	CA_VariantGetShortPtr
Variant Get Int Ptr	CA_VariantGetIntPtr
Variant Get Bool Ptr	CA_VariantGetBoolPtr
Variant Get Float Ptr	CA_VariantGetFloatPtr
Variant Get Double Ptr	CA_VariantGetDoublePtr
Variant Get Safe Array Ptr	CA_VariantGetSafeArrayPtr
Variant Get BSTR Ptr	CA_VariantGetBSTRPtr
Variant Get Currency Ptr	CA_VariantGetCurrencyPtr
Variant Get Date Ptr	CA_VariantGetDatePtr

Table 11-1. Functions in the ActiveX Automation Library Function Tree (Continued)

Class/Panel Name	Function Name
Variant-Related Functions (continued)	
Retrieving Values from Variants (continued)	
Variant Get Dispatch Ptr	`CA_VariantGetDispatchPtr`
Variant Get IUnknown Ptr	`CA_VariantGetIUnknownPtr`
Variant Get Error Ptr	`CA_VariantGetErrorPtr`
Variant Get Unsigned Char Ptr	`CA_VariantGetUCharPtr`
Variant Get Variant Ptr	`CA_VariantGetVariantPtr`
Array Functions	
C Array-to-Safe Array Conversion	
1D Array to Safe Array	`CA_Array1DToSafeArray`
2D Array to Safe Array	`CA_Array2DToSafeArray`
Safe Array-to-C Array Conversion	
Safe Array to 1D Array	`CA_SafeArrayTo1DArray`
Safe Array to 2D Array	`CA_SafeArrayTo2DArray`
Safe Array to 1D Array Buffer	`CA_SafeArrayTo1DArrayBuf`
Safe Array to 2D Array Buffer	`CA_SafeArrayTo2DArrayBuf`
Querying Safe Arrays	
Safe Array Get Number of Dims	`CA_SafeArrayGetNumDims`
Get 1D Safe Array Size	`CA_SafeArrayGet1DSize`
Get 2D Safe Array Size	`CA_SafeArrayGet2DSize`
BSTR Functions	
C String To BSTR	`CA_CStringToBSTR`
BSTR Get C String	`CA_BSTRGetCString`
BSTR Get C String In Buffer	`CA_BSTRGetCStringBuf`
BSTR Get C String Length	`CA_BSTRGetCStringLen`
Freeing Resources	
Free Memory	`CA_FreeMemory`
Clear Variant	`CA_VariantClear`
Destroy Safe Array	`CA_SafeArrayDestroy`
Discard Object Handle	`CA_DiscardObjHandle`
Free Unused Servers	`CA_FreeUnusedServers`
Error Processing	
Display Error Info	`CA_DisplayErrorInfo`
Get Automation Error String	`CA_GetAutomationErrorString`
Locales	
Set Locale	`CA_SetLocale`
Get Locale	`CA_GetLocale`
Low-level Functions	
Creating Automation Objects	
Get Active Object By Class Id	`CA_GetActiveObjectByClassId`
Get Active Object By Prog Id	`CA_GetActiveObjectByProgId`
Create Object By Class Id	`CA_CreateObjectByClassId`
Create Object By Prog Id	`CA_CreateObjectByProgId`
Load Object From File	`CA_LoadObjectFromFile`
Load Object From File By Class Id	`CA_LoadObjectFromFileByClassId`

Table 11-1. Functions in the ActiveX Automation Library Function Tree (Continued)

Class/Panel Name	Function Name
Low-level Functions (continued)	
Creating Automation Objects (continued)	
Load Object From File By Prog Id	`CA_LoadObjectFromFileByProgId`
Create ObjHandle from Dispatch	`CA_CreateObjHandleFromIDispatch`
Calling Methods and Properties	
Invoke Method	`CA_MethodInvoke`
Invoke Method (List)	`CA_MethodInvokeV`
Get Property	`CA_PropertyGet`
Set Property	`CA_PropertySet`
Set Property (List)	`CA_PropertySetV`
Set Property By Ref	`CA_PropertySetByRef`
Set Property By Ref (List)	`CA_PropertySetByRefV`
Invoke Method/Property	`CA_InvokeHelper`
Invoke Method/Property (List)	`CA_InvokeHelperV`
Get Dispatch From ObjHandle	`CA_GetDispatchFromObjHandle`

Class Descriptions

- The Variant-Related Functions class contains all the functions for assigning values to, or obtaining values from, variables or parameters declared with the `VARIANT` data type.
- The Passing Values as Variants class contains functions that allow you to pass variant parameters without declaring variant variables. You cannot pass strings or arrays using these functions because the conversions needed to store strings or arrays in variants might fail.
- The Assigning Values to Variants class contains functions that assign values to variant variables.
- The Querying the Type of a Variant class contains functions that allow you to query the data type of the value a variant holds.
- The Retrieving Values from Variants class contains functions that retrieve the values the variant parameters or variables hold.
- The Array Functions class contains functions that convert between C-style arrays and safe arrays and functions that can get the dimension and sizes of a safe array.
- The C Array-to-Safe Array Conversion class contains functions to create safe arrays from C-style arrays.
- The Safe Array-to-C Array Conversion class contains functions to create C-style arrays from safe arrays.
- The Querying Safe Arrays class contains functions to determine the number of dimensions and size of a safe array.

- The BSTR Functions class contains functions that convert between C-style strings and `BSTR` strings, which are Basic-style strings that store both text and length information.
- The Freeing Resources class contains functions to free resources that the Automation Controller Instrument Drivers or ActiveX Automation Library functions dynamically allocate.
- The Error Processing class contains functions that display error information based on error values that the ActiveX Automation Library functions or the Automation server functions return.
- The Locales class contains functions that set and get the language to use in the communication with the Automation server.
- The Low-level Functions class contains functions to create Automation server objects, invoke methods of those objects, and set and get properties of those objects.
- The Creating Automation Objects class contains functions to create Automation server objects, such as an Excel worksheet.
- The Calling Methods and Properties class contains functions to invoke methods of Automation server objects and to set and get properties of those objects.

The online help with each panel contains specific information about operating each function panel.

Using Input Variant Parameters

The ActiveX Automation Library contains two sets of functions to help you pass variant input parameters and set variants properties. The first set of functions allows you to pass values as variants without declaring `VARIANT` variables. The Passing Values as Variants function tree class includes these functions. You cannot pass strings or arrays using these functions because the conversions needed when storing strings or arrays in variants might fail.

Some server methods have optional variant parameters. You can tell the server to use a server-defined default value for an optional parameter by passing a variant with a special value. Use `CA_DefaultValueVariant` to pass a variant that contains this special value. You can use the `CA_DEFAULT_VAL` macro to refer to `CA_DefaultValueVariant`.

For strings and arrays, you must declare variables of type `VARIANT` and use the functions in the Assigning Values to Variants function tree class to store values in these variables. You must free the strings or arrays stored in variants when you no longer need them. Use `CA_VariantClear` to free the contents of variants.

Using Output Variant Parameters

The ActiveX Automation Library contains functions to query the type of value a variant contains and functions to retrieve values from a variant. `CA_VariantGetType` returns a constant that represents the type of value the variant contains. Other functions, such as

CA_VariantHasLong and CA_VariantHasShort, return a Boolean value that indicates whether the variant contains a value of a specific type.

CA_VariantConvertToType converts the value the variant contains to a type you specify. Other functions, such as CA_VariantGetLong, retrieve a value of a specific type from a variant and fail if the variant does not contain a value of that type.

Variants Marked as Empty by Retrieval Functions

All the functions that retrieve values from a variant mark the variant as empty and free any dynamically allocated memory the variant holds. Thus, you cannot call the retrieval functions multiple times on the same variant. If you do not retrieve the values from a variant, you can free the contents of a variant using CAVariantClear.

Data Types for Variants, Safe Arrays, and Properties

A set of fundamental data types exists that is valid for variants, safe arrays, and properties. You can apply a set of modifiers to the fundamental data types to create more data types. Not all combinations are valid in all cases. The function descriptions specify which data types are valid in particular contexts. Table 11-2 shows the fundamental data types.

Table 11-2. Fundamental Data Types for Variants, Safe Arrays, and Properties

Defined Constant	Data Type or Meaning
CAVT_EMPTY	Variant contains nothing.
CAVT_NULL	Variant contains NULL value.
CAVT_SHORT	short
CAVT_LONG	long
CAVT_INT	int (same as CAVT_LONG)
CAVT_FLOAT	float
CAVT_DOUBLE	double
CAVT_CY	CURRENCY (Windows SDK data type)
CAVT_DATE	DATE (Windows SDK data type)
CAVT_BSTR	BSTR (Windows SDK data type)
CAVT_DISPATCH	LPDISPATCH (ActiveX data type for an automation object interface)
CAVT_ERROR	SCODE (Windows SDK data type)

Table 11-2. Fundamental Data Types for Variants, Safe Arrays, and Properties (Continued)

Defined Constant	Data Type or Meaning
CAVT_BOOL	VBOOL, which maps to VARIANT_BOOL (ActiveX data type)
CAVT_VARIANT	VARIANT (ActiveX data type)
CAVT_UNKNOWN	LPUNKNOWN (ActiveX data type for an unknown interface)
CAVT_UCHAR	unsigned char
CAVT_CSTRING	char* (null-terminated string)
CAVT_OBJHANDLE	CAObjHandle, which maps to void*

You can bitwise OR all the constants in Table 11-2 with the data type modifiers except for CAVT_EMPTY and CAVT_NULL. Table 11-3 shows the data type modifiers.

Table 11-3. Data Types Modifiers for Variants, Safe Arrays, and Properties

Defined Constant	Meaning
CAVT_ARRAY	Array of data type. For example, CAVT_SHORT|CAVT_ARRAY signifies an array of short integers.
CAVT_BYREF	Pointer to data type.
CAVT_BYREFI	Pointer to data type. Input-only parameter you pass by reference. Defined as CAVT_BYREF|CAVT_IN.
CAVT_BYREFO	Pointer to data type. Output-only parameter you pass by reference. Defined as CAVT_BYREF|CAVT_OUT.
CAVT_BYREFIO	Pointer to data type. Input-output parameter you pass by reference. Defined as CAVT_BYREF|CAVT_IN|CAVT_OUT.

To pass a pointer to the array address, use CAVT_ARRAY and one of the CAVT_BYREF modifiers.

Handling Dynamic Memory Variants Hold

Variants of the following data types store their values in dynamically allocated memory:

```
CAVT_CSTRING
CAVT_OBJHANDLE
<any type> | CAVT_ARRAY
```

The functions that retrieve values from such variants free the memory. If you do not retrieve the value stored in the variant, you can free the contents of a variant using `CA_VariantClear`.

ActiveX Automation Library Function Reference

This section describes each function in the LabWindows/CVI ActiveX Automation Library in alphabetical order.

CA_Array1DToSafeArray

```
HRESULT status = CA_Array1DToSafeArray (void *array, unsigned int arrayType,
                                        unsigned int numElements,
                                        LPSAFEARRAY *safeArray);
```

Purpose

Creates a safe array from a 1D array.

Parameters

Input

Name	Type	Description
array	void pointer	1D array.
arrayType	unsigned integer	Data type of the elements in **array**.
numElements	unsigned integer	Number of elements in **array**.

Output

Name	Type	Description
safeArray	`LPSAFEARRAY`	Safe array `CA_Array1DToSafeArray` creates from the contents of **array**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter can have any of the values in Table 11-2 except for `CAVT_EMPTY`, `CAVT_NULL`, or `CAVT_OBJHANDLE`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

`CA_Array1DToSafeArray` does not make copies of `BSTR`, `VARIANT`, `LPUNKNOWN`, or `LPDISPATCH` elements. It simply copies the pointers from **array** into the created **safeArray**. Therefore, when you call `CA_SafeArrayDestroy`,which frees all contents of the safe array, the `BSTR`, `VARIANT`, `LPUNKNOWN`, or `LPDISPATCH` elements in **array** become invalid.

CA_Array2DToSafeArray

```
HRESULT status = CA_Array2DToSafeArray (void *array, unsigned int arrayType,
                                        unsigned int numElemsDim1,
                                        unsigned int numElemsDim2,
                                        LPSAFEARRAY *safeArray);
```

Purpose

Creates a safe array from a 2D array.

Parameters

Input

Name	Type	Description
array	void pointer	2D array.
arrayType	unsigned integer	Data type of the elements in **array**.
numElemsDim1	unsigned integer	Number of rows in **array**.
numElemsDim2	unsigned integer	Number of columns in **array**.

Output

Name	Type	Description
safeArray	`LPSAFEARRAY`	Safe array `CA_Array2DToSafeArray` creates from the contents of **array**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter can have any of the values in Table 11-2 except for `CAVT_EMPTY`, `CAVT_NULL`, or `CAVT_OBJHANDLE`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

`CA_Array2DToSafeArray` does not make copies of `BSTR`, `VARIANT`, `LPUNKNOWN`, or `LPDISPATCH` elements. It simply copies the pointers from **array** into the created **safeArray**. Therefore, when you call `CA_SafeArrayDestroy`, which frees all contents of the safe array, the `BSTR`, `VARIANT`, `LPUNKNOWN`, or `LPDISPATCH` elements in **array** become invalid.

CA_BSTRGetCString

```
HRESULT status = CA_BSTRGetCString (BSTR bString, char **cString);
```

Purpose

Converts a BSTR into a C-style string. A BSTR is a Basic-style string that stores both text and length information.

CA_BSTRGetCString does not free the Basic-style string.

Parameters

Input

Name	Type	Description
bString	BSTR	Basic-style string you want to convert to a C-style string.

Output

Name	Type	Description
cString	string	Dynamically allocated C-style string that CA_BSTRGetCString converts from the Basic-style string.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When you no longer need the C-style string, call CA_FreeMemory to free it.

CA_BSTRGetCStringBuf

```
HRESULT status = CA_BSTRGetCStringBuf (BSTR bString, char buffer[],
                                       unsigned long bufferSize);
```

Purpose

Converts a BSTR into a C-style string and copies it into a buffer you pass as a parameter. A BSTR is a Basic-style string that stores both text and length information.

CA_BSTRGetCStringBuf does not free the Basic-style string.

If **buffer** is not large enough to hold the string, CA_BSTRGetCStringBuf copies (**bufferSize** – 1) bytes to the **buffer**, followed by an ASCII NUL byte.

Parameters

Input

Name	Type	Description
bString	BSTR	Basic-style string you want to convert to a C-style string.
bufferSize	unsigned long integer	Number of bytes in **buffer**. Must be must be large enough to hold the string text and an ASCII NUL byte.

Output

Name	Type	Description
buffer	string	Character buffer into which the CA_BSTRGetCString copies the C-style string it converts from the Basic-style string.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_BSTRGetCStringLen

```
HRESULT status = CA_BSTRGetCStringLen (BSTR bString, int *len);
```

Purpose

Obtains the length of the C-style string you can create by calling `CA_BSTRToCString` on a `BSTR` you specify. A `BSTR` is a Basic-style string that stores both text and length information.

The length `CA_BSTRGetCStringLen` returns does not include the ASCII NUL byte.

Parameters

Input

Name	Type	Description
bString	`BSTR`	Basic-style string.

Output

Name	Type	Description
len	integer	Length of the C-style string `CA_BSTRToCString` can convert from the Basic-style string. Does not include the ASCII NUL byte.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_CreateObjectByClassId

```
HRESULT status = CA_CreateObjectByClassId (GUID *classID,
                          char *serverMachineName,
                          CAObjHandle *objHandle);
```

Purpose

Creates a new Automation server object based on an object Class ID.

If the application that provides the Automation object is already running, `CA_CreateObjectByClassId` might start another copy of the application, depending on the implementation of the application.

Parameters

Input

Name	Type	Description
classID	`GUID` pointer	Class ID of the Automation server object; located in the server type library.
serverMachineName	string	Name or IP address of the computer on which you want to run the Automation server.

Output

Name	Type	Description
objHandle	`CAOHandle`	Handle to the server object you create.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

The **serverMachineName** can be either a UNC name ("\\server") or a DNS name ("home.server.com").

If you pass NULL for the **serverMachineName** and a RemoteServerName registry entry exists for this server, the server runs on the computer the RemoteServerName entry specifies. If you pass NULL for this parameter and no RemoteServerName registry entry exists for this server, the server runs on the same computer as your program.

You can pass **objHandle** to other functions in this library to call methods of the Automation object or to get and set properties of the Automation object. When you no longer need **objHandle**, discard it by calling CA_DiscardObjHandle.

CA_CreateObjectByProgId

```
HRESULT status = CA_CreateObjectByProgId (char *progID,
                          char *serverMachineName, CAObjHandle *objHandle);
```

Purpose

Creates a new Automation server object based on the object Prog ID.

If the application that provides the Automation object is already running, `CA_CreateObjectByProgId` might start another copy of the application, depending on the implementation of the application.

Parameters

Input

Name	Type	Description
progID	string	Prog ID of the Automation server object; located in the server documentation.
serverMachineName	string	Name or IP address of the computer on which you want to run the Automation server.

Output

Name	Type	Description
objHandle	`CAObjHandle`	Handle to the server object you create.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **serverMachineName** can be either a UNC name (`"\\server"`) or a DNS name (`"home.server.com"`).

If you pass `NULL` for the **serverMachineName** and a `RemoteServerName` registry entry exists for this server, the server runs on the computer the `RemoteServerName` entry specifies. If you pass `NULL` for this parameter and no `RemoteServerName` registry entry exists for this server, the server runs on the same computer as your program.

You can pass **objHandle** to other functions in this library to call methods of the Automation object or to get and set properties of the Automation object. When you no longer need **objHandle**, discard it by calling `CA_DiscardObjHandle`.

CA_CreateObjHandleFromIDispatch

```
HRESULT status = CA_CreateObjHandleFromIDispatch (LPDISPATCH dispatchPtr,
                          int callAddRef, CAObjHandle *objHandle);
```

Purpose

Creates a `CAObjHandle` value when you already have a Dispatch pointer to an automation object.

Parameters

Input

Name	Type	Description
dispatchPtr	`LPDISPATCH`	Dispatch pointer to an Automation server object.
callAddRef	integer	If nonzero, `CA_CreateObjHandleFromIDispatch` invokes the `AddRef` method of the object.

Output

Name	Type	Description
objHandle	`CAObjHandle`	Handle to the server object you create.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

You can pass **objHandle** to other functions in this library to call methods of the Automation object or to get and set properties of the Automation object. When you no longer need **objHandle**, discard it by calling `CA_DiscardObjHandle`.

CA_CStringToBSTR

```
HRESULT status = CA_CStringToBSTR (char cString[], BSTR *bString);
```

Purpose

Converts a C-style string into a BSTR. A BSTR is a Basic-style string that stores both text and length information.

CA_CStringToBSTR does not free the C-style string.

Parameters

Input

Name	Type	Description
cString	string	C-style string you want to convert to a BSTR.

Output

Name	Type	Description
bString	BSTR	Basic-style string CA_CSringToBSTR creates from **cString**.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When you no longer need the Basic-style string, call the Windows SDK function SysFreeString to free it.

CA_DefaultValueVariant

```
VARIANT *variant = CA_DefaultValueVariant (void);
```

Purpose

Some Automation server functions allow you to pass a VARIANT parameter value that means "use the default value for this parameter." CA_DefaultValueVariant returns such a variant. You can use the macro CA_DEFAULT_VAL to refer to CA_DefaultValueVariant.

You can use the returned variant only for parameters that are optional. You can tell whether a parameter is optional by checking the default value in its function panel control. If the default value is CA_DEFAULT_VAL, the parameter is optional.

CA_DefaultValueVariant sets the type field in the returned variant to VT_ERROR and the error value field to DISP_E_PARAMNOTFOUND.

Parameters

None.

Return Value

Name	Type	Description
variant	VARIANT	Variant in which LabWindows/CVI sets the type field to VT_ERROR and the error value to DISP_E_PARAMNOTFOUND.

CA_DiscardObjHandle

```
HRESULT status = CA_DiscardObjHandle (CAObjHandle objHandle);
```

Purpose

Use `CA_DiscardObjHandle` when you no longer need to reference an Automation server object. `CA_DiscardObjHandle` releases resources associated with the object and calls the `Release` method of the Automation server object.

If **objHandle** is the only reference to the Automation server and the Automation server is an application, the server application might shut down after you call `CA_DiscardObjHandle`. If you implement the Automation server as a DLL, you must call the Windows SDK function `CoFreeUnusedLibraries` to unload the DLL. If you do not call `CoFreeUnusedLibraries`, the DLL does not unload until you exit your program.

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_DisplayErrorInfo

```
HRESULT status = CA_DisplayErrorInfo (CAObjHandle objHandle, char *title,
                                      HRESULT errorCode, ERRORINFO *errorInfo);
```

Purpose

Displays in a dialog box the description associated with an error code and the error information an `ERRORINFO` structure contains. `CA_DisplayErrorInfo` formats and displays the `sCode`, `wCode`, `source`, `description`, and `errorParamPos` fields of the `ERROINFO` structure, unless they are `NULL`.

If the `helpFile` of the `ERROINFO` structure field is not `NULL`, the dialog box includes a **Help** button. To display the help file, click on the **Help** button.

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	Object handle you pass to the Automation server function or ActiveX Automation Library function that reported the error.
title	string	Title of the dialog box. If you pass `NULL`, `Automation Error` appears as the title.
errorCode	`HRESULT`	Error code an Automation server function or an ActiveX Automation Library function returns.
errorInfo	`ERRORINFO`	Structure the ActiveX Automation Library fills in when a server method fails. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_FreeMemory

```
void CA_FreeMemory (void *memPtr);
```

Purpose

Frees memory allocated through the following sources:

- String output parameters of functions in an Automation Instrument Driver
- Strings that the following functions allocate: `CA_VariantGetCString`, `CA_BSTRGetCString`, and `CA_VariantConvertToType`
- Arrays that the following functions allocate: `CA_SafeArrayTo1DArray`, `CA_SafeArrayTo2DArray`, `CA_VariantGet1DArray`, and `CA_VariantGet2DArray`
- String array elements of arrays that the following functions allocate: `CA_SafeArrayTo1DArray`, `CA_SafeArrayTo2DArray`, `CA_SafeArrayTo1DArrayBuf`, `CA_SafeArrayTo2DArrayBuf`, `CA_VariantGet1DArray`, `CA_VariantGet2DArray`, `CA_VariantGet1DArrayBuf`, and `CA_VariantGet2DArrayBuf`

Parameters

Input

Name	Type	Description
memPtr	void pointer	Address of the memory to free.

Return Value

None.

CA_FreeUnusedServers

```
void FreeUnusedServers(void)
```

Purpose

Unloads ActiveX Automation server DLLs that you are no longer using.

For each server that is in DLL form, `CA_UnloadUnusedServers` unloads the server DLL if you have already called `CA_DiscardObjectHandle` on all object handles for that server. This includes object handles that you create by calling server methods, functions in the ActiveX Automation Library, and functions in generated automation controller instrument drivers that return object handles, such as New, Open, and Active.

This function has no effect on servers that are running in separate processes.

Parameters

None.

Return Value

None.

CA_GetActiveObjectByClassId

```
HRESULT status = CA_GetActiveObjectByClassId (GUID *classID,
                                char *serverMachineName, CAObjHandle *objHandle);
```

Purpose

Obtains a handle to an active Automation server object based on the Class ID of the object.

Parameters

Input

Name	Type	Description
classID	`GUID` pointer	Class ID of the Automation server object; located in the server type library.
serverMachineName	string	Name or IP address of the computer on which you want to run the Automation server.

Output

Name	Type	Description
objHandle	`CAObjHandle`	Handle to the server object you create.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **serverMachineName** can be either a UNC name (`"\\server"`) or a DNS name (`"home.server.com"`). If you pass `NULL` for the **serverMachineName**, `CA_GetActiveObjectByClassId` looks for the active Automation server on the same computer as your program.

Note ***Windows 95 and Windows NT 4.0 do not allow you to access active objects on remote machines. Future versions of these operating systems might support this functionality.***

You can pass **objHandle** to other functions in this library to call methods of the Automation object or to get and set properties of the Automation object. When you no longer need **objHandle**, discard it by calling `CA_DiscardObjHandle`.

CA_GetActiveObjectByProgId

```
HRESULT status = CA_GetActiveObjectByProgId (char *progID,
                                char *serverMachineName, CAObjHandle *objHandle);
```

Purpose

Obtains a handle to an active Automation server object based on the Prog ID of the object.

Parameters

Input

Name	Type	Description
progID	string	Prog ID of the Automation server object; located in the server documentation.
serverMachineName	string	Name or IP address of the computer on which you want to run the Automation server.

Output

Name	Type	Description
objHandle	`CAObjHandle`	Handle to the server object you create.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **serverMachineName** can be either a UNC name (`"\\server"`) or a DNS name (`"home.server.com"`). If you pass `NULL` for the **serverMachineName**, `CA_GetActiveObjectByProgId` looks for the active Automation server on the same computer as your program.

Note ***Windows 95 and Windows NT 4.0 do not allow you to access active objects on remote machines. Future versions of these operating systems might support this functionality.***

You can pass **objHandle** to other functions in this library to call methods of the Automation object or to get and set properties of the Automation object. When you no longer need **objHandle**, discard it by calling `CA_DiscardObjHandle`.

CA_GetAutomationErrorString

```
void CA_GetAutomationErrorString (HRESULT errorCode, char buffer[],
                                  unsigned int bufferSize);
```

Purpose

Converts an error code number into a meaningful error string.

Parameters

Input

Name	Type	Description
errorCode	`HRESULT`	Error code an Automation server function or an ActiveX Automation Library function returns.

Output

Name	Type	Description
buffer	character array	Buffer into which `CA_GetAutomationErrorString` copies the error string.
bufferSize	unsigned integer	Number of bytes in **buffer**.

Return Value

None.

Parameter Discussion

If **buffer** cannot hold the entire error string, `CA_GetAutomationErrorString` copies (**bufferSize**−1) bytes into **buffer**, appended by the ASCII NUL byte.

CA_GetDispatchFromObjHandle

```
HRESULT status = CA_GetDispatchFromObjHandle (CAObjHandle objHandle,
                                              LPDISPATCH *dispatchPtr);
```

Purpose

Obtains the Dispatch pointer associated with the `CAObjHandle` for an Automation server object. You can use the Dispatch pointer to call members of the server IDispatch interface, or you can pass the Dispatch pointer to Windows SDK functions.

`CA_GetDispatchFromObjHandle` does not invoke the `AddRef` method on the Dispatch pointer.

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.

Output

Name	Type	Description
dispatchPtr	`LPDISPATCH`	Dispatch pointer of the Automation server object **objHandle** identifies.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_GetLocale

```
HRESULT status = CA_GetLocale (CAObjHandle objHandle, LCID *locale);
```

Purpose

Obtains the language the Automation server uses to interpret the arguments it receives as parameters to its functions.

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.

Output

Name	Type	Description
locale	`LCID`	ID that indicates the language the Automation server uses. Refer to `CA_SetLocale` for a list of locale IDs.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_InvokeHelper

```
HRESULT status = CA_InvokeHelper (CAObjHandle objHandle,
                          ERRORINFO *errorInfo, int methodOrPropertyID,
                          int operation, unsigned int returnType,
                          void *returnValue, int parameterCount,
                          unsigned int parameterTypes[], ...);
```

Purpose

Gets or sets the value of an Automation server property or invokes a method of an Automation server. Unlike `CA_InvokeHelperV`, `CA_InvokeHelper` accepts arguments to the server operation as comma-separated parameters.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_InvokeHelper`***. You do not need to call it directly.***

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.
methodOrPropertyID	integer	ID of the method or property you call or access; located in the server type library.
operation	integer	Type of operation you want the server to perform.
returnType	unsigned integer	Type of the return value (if any).
parameterCount	integer	Number of arguments you pass that follow **parameterTypes**.
parameterTypes	unsigned integer array	Data types of each argument that follows this parameter.
parameters	depends on the values in **parameterTypes**	Arguments to the Automation server operation. You must separate multiple arguments with commas.

Output

Name	Type	Description
errorInfo	ERRORINFO	Structure CA_InvokeHelper fills in when a server method fails. You can pass NULL.
returnValue	void pointer	The value (if any) the server method or property function returns.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When an Automation server method invoked by CA_InvokeHelper fails with the error code DISP_E_EXCEPTION, CA_InvokeHelper stores descriptive information about the error in the **errorInfo** parameter. The descriptive information includes the error code, source, and description. It also can include a help file and help file context string.

When an Automation server method invoked by CA_InvokeHelper fails with the error codes DISP_E_PARAMNOTFOUND, DISP_E_TYPEMISMATCH, or E_INVALIDARG, CA_InvokeHelper might store the parameter position of the invalid argument in the errorParamPos member of the **errorInfo** structure.

You can pass NULL for the **errorInfo** parameter.

The **operation** parameter must be one of the values in Table 11-4.

Table 11-4. operation Parameter Values

Defined Constant	Type of Operation
DISPATCH_METHOD	Invoke a method.
DISPATCH_PROPERTYGET	Get a property value.
DISPATCH_PROPERTYPUT	Set a property value.
DISPATCH_PROPERTYPUTREF	Set a property value (pass a pointer to the value).

The value you pass for **returnType** depends on the **operation** parameter as shown in Table 11-5.

Table 11-5. Return Type Values

Operation	Return Type
DISPATCH_METHOD	Return type of the server function.
DISPATCH_PROPERTYGET	Data type of the property.
DISPATCH_PROPERTYPUT	Always CAVT_EMPTY.
DISPATCH_PROPERTYPUTREF	Always CAVT_EMPTY.

For DISPATCH_METHOD and DISPATCH_PROPERTYGET, the **returnType** can be any of the types in Table 11-2 in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter except CAVT_NULL. You can use the CAVT_ARRAY modifier on all data types except CAVT_EMPTY, CAVT_CSTRING, and CAVT_OBJHANDLE.

The value you pass for **parameterCount** depends on the **operation** parameter as shown in Table 11-6.

Table 11-6. Parameter Count Values

Operation	Parameter Count
DISPATCH_METHOD	Number of arguments to pass to the server function.
DISPATCH_PROPERTYGET	0
DISPATCH_PROPERTYPUT	1
DISPATCH_PROPERTYPUTREF	1

The value you pass for **parameterTypes** depends on the **operation** parameter as shown in Table 11-7.

Table 11-7. Parameter Types Values

Operation	Parameter Types
DISPATCH_METHOD	Array that contains the types of the arguments to the server function.
DISPATCH_PROPERTYGET	NULL
DISPATCH_PROPERTYPUT	Single-element array that contains the data type of the property.
DISPATCH_PROPERTYPUTREF	Single-element array that contains the data type of the property.

The **parameterTypes** array can contain any of the data types in Table 11-2 except CAVT_NULL or CAVT_EMPTY. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. You can use the CAVT_ARRAY modifier on all data types except CAVT_CSTRING and CAVT_OBJHANDLE. For parameters you pass by reference using DISPATCH_METHOD, you can use the CAVT_BYREFI, CAVT_BYREFO, or CAVT_BYREFIO modifier. For DISPATCH_PROPERTYPUTREF, you should add the CAVT_BYREFI modifier, but the ActiveX Automation Library adds it for you if you forget.

The values you pass for **parameters** depends on the **operation** parameter as shown in Table 11-8.

Table 11-8. Parameter Values

Operation	Parameters
DISPATCH_METHOD	Arguments to the server function. You must separate multiple arguments with commas.
DISPATCH_PROPERTYGET	Do not pass any arguments.
DISPATCH_PROPERTYPUT	Value to which you want to set the property.
DISPATCH_PROPERTYPUTREF	Pointer to the value to which you want to set the property.

The value `CA_InvokeHelper` returns in **returnValue** depends on the **operation** parameter as shown in Table 11-9.

Table 11-9. Return Values

Operation	Return Value
`DISPATCH_METHOD`	Value the server function returns. Pass a pointer to a variable of the data type **returnType** specifies.
`DISPATCH_PROPERTYGET`	Value of the property. Pass a pointer to a variable of the data type **returnType** specifies.
`DISPATCH_PROPERTYPUT`	None. Always pass `NULL`.
`DISPATCH_PROPERTYPUTREF`	None. Always pass `NULL`.

CA_InvokeHelperV

```
HRESULT status = CA_InvokeHelperV (CAObjHandle objHandle,
                           ERRORINFO *errorInfo, int methodOrPropertyID,
                           int operation, unsigned int returnType,
                           void *returnValue, int parameterCount,
                           unsigned int parameterTypes[],
                           va_list parameters);
```

Purpose

Gets or sets the value of an Automation server property or invokes a method of an Automation server. Unlike `CA_InvokeHelper`, `CA_InvokeHelperV` accepts arguments to the server operation as a variable argument list.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_InvokeHelperV`***. You do not need to call it directly.***

Parameters

The parameters to `CA_InvokeHelperV` are the same as the parameters to `CA_InvokeHelper` except that you must pass the parameters to the Automation server operation as a variable argument list (`va_list`) that you initialize with the `va_start` macro.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_LoadObjectFromFile

```
HRESULT status = CA_LoadObjectFromFile (char *filename,
                                        char *serverMachineName, CAObjHandle *objHandle);
```

Purpose

Creates an Automation server object and initializes it using data `CA_LoadObjectFromFile` reads from a file. The extension portion of the **filename** parameter identifies the Automation server to use. The contents of the file identifies the type of object to create and its initial data.

Parameters

Input

Name	Type	Description
filename	string	Pathname of file that contains the type of object to create and its initial data. The extension indicates the Automation server to use.
serverMachineName	string	Name or IP address of the computer on which you want to run the Automation server.

Output

Name	Type	Description
objHandle	`CAObjHandle`	Handle to the server object you create.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **serverMachineName** can be either a UNC name (`"\\server"`) or a DNS name (`"home.server.com"`).

If you pass `NULL` for the **serverMachineName** and a `RemoteServerName` registry entry exists for this server, the server runs on the computer the `RemoteServerName` entry specifies. If you pass `NULL` for this parameter and no `RemoteServerName` registry entry exists for this server, the server runs on the same computer as your program.

You can pass **objHandle** to other functions in this library to call methods of the Automation object or to get and set properties of the Automation object. When you no longer need **objHandle**, discard it by calling `CA_DiscardObjHandle`.

CA_LoadObjectFromFileByClassId

```
HRESULT status = CA_LoadObjectFromFileByClassId (char *filename,
                          GUID *classID, char *serverMachineName,
                          CAObjHandle *objHandle);
```

Purpose

Creates an Automation server object and initializes it using data `CA_LoadObjectFromFileByClassId` reads from a file. The **classID** parameter identifies the Automation server object. **filename** specifies the file that contains the initial data.

Parameters

Input

Name	Type	Description
filename	string	Pathname of file that contains the initial data for the object.
classID	`GUID` pointer	Class ID of the Automation server object; located in the server type library.
serverMachineName	string	Name or IP address of the computer on which you want to run the Automation server.

Output

Name	Type	Description
objHandle	`CAObjHandle`	Handle to the server object you create.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **serverMachineName** can be either a UNC name (`"\\server"`) or a DNS name (`"home.server.com"`).

If you pass `NULL` for the **serverMachineName** and a `RemoteServerName` registry entry exists for this server, the server runs on the computer the `RemoteServerName` entry specifies. If you pass `NULL` for this parameter and no `RemoteServerName` registry entry exists for this server, the server runs on the same computer as your program.

You can pass **objHandle** to other functions in this library to call methods of the Automation object or to get and set properties of the Automation object. When you no longer need **objHandle**, discard it by calling `CA_DiscardObjHandle`.

CA_LoadObjectFromFileByProgId

```
HRESULT status = CA_LoadObjectFromFileByProgId (char *filename,
                          char *progID, char *serverMachineName,
                          CAObjHandle *objHandle);
```

Purpose

Creates an Automation server object and initializes it using data `CA_LoadObjectFromFileByProgId` reads from a file. The **progID** parameter identifies the Automation server object. **filename** specifies the file that contains the initial data.

Parameters

Input

Name	Type	Description
filename	string	Pathname of file that contains the initial data for the object.
progID	string	ProgID of the Automation server object; located in the server documentation.
serverMachineName	string	Name or IP address of the computer on which you want to run the Automation server.

Output

Name	Type	Description
objHandle	`CAObjHandle`	Handle to the server object you create.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **serverMachineName** can be either a UNC name (`"\\server"`) or a DNS name (`"home.server.com"`).

If you pass `NULL` for the **serverMachineName** and a `RemoteServerName` registry entry exists for this server, the server runs on the computer the `RemoteServerName` entry specifies. If you pass `NULL` for this parameter and no `RemoteServerName` registry entry exists for this server, the server runs on the same computer as your program.

You can pass **objHandle** to other functions in this library to call methods of the Automation object or to get and set properties of the Automation object. When you no longer need **objHandle**, discard it by calling `CA_DiscardObjHandle`.

CA_MethodInvoke

```
HRESULT status = CA_MethodInvoke (CAObjHandle objHandle,
                          ERRORINFO *errorInfo, int methodID,
                          unsigned int returnType, void *returnValue,
                          int parameterCount,
                          unsigned int parameterTypes[], ...);
```

Purpose

Invokes an Automation server method. Unlike `CA_MethodInvokeV`, `CA_MethodInvoke` accepts arguments to the server operation as comma-separated parameters.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_MethodInvoke`***. You do not need to call it directly.***

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.
methodID	integer	ID of the method you call; located in the server type library.
returnType	unsigned integer	Data type of the return value of the server method.
parameterCount	integer	Number of arguments you pass that follow **parameterTypes**.
parameterTypes	unsigned integer array	Data types of each argument that follows this parameter.
parameters	depends on the values in **parameterTypes**	Arguments to the Automation server method. You must separate multiple arguments with commas.

Output

Name	Type	Description
errorInfo	ERRORINFO	Structure CA_MethodInvoke fills in when a server method fails. You can pass NULL.
returnValue	void pointer	The value the server method returns. Pass a pointer to a variable of the data type **returnType** specifies.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When an Automation server method invoked by CA_MethodInvoke fails with the error code DISP_E_EXCEPTION, CA_MethodInvoke stores descriptive information about the error in the **errorInfo** parameter. The descriptive information includes the error code, source, and description. It also can include a help file and help file context string.

When an Automation server method invoked by CA_MethodInvoke fails with the error codes DISP_E_PARAMNOTFOUND, DISP_E_TYPEMISMATCH, or E_INVALIDARG, CA_MethodInvoke might store the parameter position of the invalid argument in the errorParamPos member of the **errorInfo** structure.

You can pass NULL for the **errorInfo** parameter.

The **returnType** can be any of the types in Table 11-2 except CAVT_NULL. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. You can use the CAVT_ARRAY modifier on all data types except CAVT_EMPTY, CAVT_CSTRING, and CAVT_OBJHANDLE.

The **parameterTypes** array can contain any of the data types in Table 11-2 except CAVT_NULL or CAVT_EMPTY. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. You can use the CAVT_ARRAY modifier on all data types except CAVT_CSTRING and CAVT_OBJHANDLE. For parameters you pass by reference, you can use the CAVT_BYREFI, CAVT_BYREFO, or CAVT_BYREFIO modifier.

CA_MethodInvokeV

```
HRESULT status = CA_MethodInvokeV (CAObjHandle objHandle,
                           ERRORINFO *errorInfo, int methodID,
                           unsigned int returnType, void *returnValue,
                           int parameterCount,
                           unsigned int parameterTypes[],
                           va_list parameterList);
```

Purpose

Invokes a method of an Automation server. Unlike `CA_MethodInvoke`, `CA_MethodInvokeV` accepts arguments to the server operation as a variable argument list.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_MethodInvokeV`***. You do not need to call it directly.***

Parameters

The parameters to `CA_MethodInvokeV` are the same as the parameters to `CA_MethodInvoke` except that you must pass the parameters to the Automation server method as a variable argument list (`va_list`) that you initialize with the `va_start` macro.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_PropertyGet

```
HRESULT status = CA_PropertyGet (CAObjHandle objHandle,
                                 ERRORINFO *errorInfo, int propertyID,
                                 unsigned int propertyType, void *propertyValue);
```

Purpose

Obtains the value of the property of an Automation server object.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_PropertyGet`***. You do not need to call it directly.***

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.
propertyID	integer	ID of the automation server property; located in the server type library.
propertyType	unsigned integer	Data type of the property.

Output

Name	Type	Description
errorInfo	`ERRORINFO`	Structure `CA_PropertyGet` fills in when a server function fails. You can pass `NULL`.
propertyValue	depends on the value of **propertyType**	Value the server method returns. Pass a pointer to a variable of the data type **propertyType** specifies.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

When an Automation server method invoked by `CA_PropertyGet` fails with the error code `DISP_E_EXCEPTION`, `CA_PropertyGet` stores descriptive information about the error in the **errorInfo** parameter. The descriptive information includes the error code, source, and description. It also can include a help file and help file context string.

When an Automation server method invoked by `CA_PropertyGet` fails with the error codes `DISP_E_PARAMNOTFOUND`, `DISP_E_TYPEMISMATCH`, or `E_INVALIDARG`, `CA_PropertyGet` might store the parameter position of the invalid argument in the `errorParamPos` member of the **errorInfo** structure.

You can pass `NULL` for the **errorInfo** parameter.

The **propertyType** can be any of the data types in Table 11-2 except `CAVT_NULL` or `CAVT_EMPTY`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. You can use the `CAVT_ARRAY` modifier on all data types except `CAVT_CSTRING` and `CAVT_OBJHANDLE`.

CA_PropertySet

```
HRESULT status = CA_PropertySet (CAObjHandle objHandle,
                                 ERRORINFO *errorInfo, int propertyID,
                                 unsigned int propertyType, ...);
```

Purpose

Sets the value of the property of an Automation server object. Unlike `CA_SetPropertyV`, `CA_PropertySet` accepts the property value as a simple parameter.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_PropertySet`***. You do not need to call it directly.***

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.
propertyID	integer	ID of the automation server property; located in the server type library.
propertyType	unsigned integer	Data type of the property.
propertyValue	depends on the value of **propertyType**	Value to which you want to set the property. Pass a value with the data type **propertyType** specifies. If **propertyType** is `VARIANT`, you can pass a value of any type.

Output

Name	Type	Description
errorInfo	`ERRORINFO`	Structure `CA_PropertySet` fills in when a server function fails. You can pass `NULL`.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When an Automation server method invoked by CA_PropertySet fails with the error code DISP_E_EXCEPTION, CA_PropertySet stores descriptive information about the error in the **errorInfo** parameter. The descriptive information includes the error code, source, and description. It also can include a help file and help file context string.

When an Automation server method invoked by CA_PropertySet fails with the error codes DISP_E_PARAMNOTFOUND, DISP_E_TYPEMISMATCH, or E_INVALIDARG, CA_PropertySet might store the parameter position of the invalid argument in the errorParamPos member of the **errorInfo** structure.

The **propertyType** can be any of the data types in Table 11-2 except CAVT_NULL or CAVT_EMPTY. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. You can use the CAVT_ARRAY modifier on all data types except CAVT_CSTRING and CAVT_OBJHANDLE.

CA_PropertySetByRef

```
HRESULT status = CA_PropertySetByRef (CAObjHandle objHandle,
                          ERRORINFO *errorInfo, int propertyID,
                          unsigned int propertyType, ...);
```

Purpose

Sets the value of the property of an Automation server object. Unlike `CA_SetProperty`, `CA_PropertySetByRef` accepts a pointer to the property value. Unlike `CA_SetPropertyByRefV`, it accepts the pointer as a simple parameter.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_PropertySetByRef`***. You do not need to call it directly.***

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.
propertyID	integer	ID of the automation server property; located in the server type library.
propertyType	unsigned integer	Data type of the property.
pointerToValue	pointer to the type **propertyType** specifies	Pointer to the value to which you want to set the property. Must point to the data type **propertyType** specifies. If **propertyType** is `VARIANT`, you can pass a pointer to any type. If the **propertyType** includes the `CAVT_ARRAY` modifier, you must pass the address of a pointer to an array.

Output

Name	Type	Description
errorInfo	`ERRORINFO`	Structure `CA_PropertySetByRef` fills in when a server function fails. You can pass `NULL`.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When an Automation server method invoked by CA_PropertySetByRef fails with the error code DISP_E_EXCEPTION, CA_PropertySetByRef stores descriptive information about the error in the **errorInfo** parameter. The descriptive information includes the error code, source, and description. It also can include a help file and help file context string.

When an Automation server method invoked by CA_PropertySetByRef fails with the error codes DISP_E_PARAMNOTFOUND, DISP_E_TYPEMISMATCH, or E_INVALIDARG, CA_PropertySetByRef might store the parameter position of the invalid argument in the errorParamPos member of the **errorInfo** structure.

The **propertyType** can be any of the data types in Table 11-2 except CAVT_NULL or CAVT_EMPTY. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. You can use the CAVT_ARRAY modifier on all data types except CAVT_CSTRING and CAVT_OBJHANDLE. You should add the CAVT_BYREFI modifier, but the ActiveX Automation Library adds it for you if you forget.

CA_PropertySetByRefV

```
HRESULT status = CA_PropertySetByRefV (CAObjHandle objHandle,
                           ERRORINFO *errorInfo, int propertyID,
                           unsigned int propertyType,
                           va_list pointerToValue);
```

Purpose

Sets the value of the property of an Automation server object. Unlike `CA_SetPropertyV`, `CA_PropertySetByRefV` accepts a pointer to the property value. Unlike `CA_SetPropertyByRef`, it accepts the pointer as the single element in a variable argument list.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_PropertySetByRefV`***. You do not need to call it directly.***

Parameters

The parameters to `CA_PropertySetByRefV` are the same as the parameters to `CA_PropertySetByRef` except that you pass **pointerToValue** as a variable argument list (`va_list`) that you initialize with the `va_start` macro.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_PropertySetV

```
HRESULT status = CA_PropertySetV (CAObjHandle objHandle,
                          ERRORINFO *errorInfo, int propertyID,
                          unsigned int propertyType,
                          va_list propertyValue);
```

Purpose

Sets the value of the property of an Automation server object. Unlike `CA_SetProperty`, `CA_PropertySetV` accepts the property value as the single element in a variable argument list.

Note ***Automation Controller Instrument Drivers you generate with the ActiveX Automation Controller Wizard use*** `CA_PropertySetV`***. You do not need to call it directly.***

Parameters

The parameters to `CA_PropertySetV` are the same as the parameters to `CA_PropertySet` except that you pass **propertyValue** as a variable argument list (`va_list`) that you initialize with the `va_start` macro.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_SafeArrayDestroy

```
HRESULT status = CA_SafeArrayDestroy (LPSAFEARRAY safeArray);
```

Purpose

Frees the memory a safe array uses.

Note ***Do not call*** `CA_SafeArrayDestroy` ***on a safe array if you pass the safe array to one of the Safe Array-to-C Array conversion functions. The Safe Array-to-C Array conversion functions free the safe array.***

Parameter

Input

Name	Type	Description
safeArray	LPSAFEARRAY	Safe array to free.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_SafeArrayGet1DSize

```
HRESULT status = CA_SafeArrayGet1DSize (LPSAFEARRAY safeArray,
                                        unsigned int *numElements);
```

Purpose

Obtains the number of elements in a 1D safe array.

Parameters

Input

Name	Type	Description
safeArray	LPSAFEARRAY	1D safe array.

Output

Name	Type	Description
numElements	unsigned integer	Number of elements in the safe array.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_SafeArrayGet2DSize

```
HRESULT status = CA_SafeArrayGet2DSize (LPSAFEARRAY safeArray,
                          unsigned int *numElemsDim1,
                          unsigned int *numElemsDim2);
```

Purpose

Obtains the number of elements in a 2D safe array.

Parameters

Input

Name	Type	Description
safeArray	LPSAFEARRAY	2D safe array.

Output

Name	Type	Description
numElemsDim1	unsigned integer	Number of elements in the first dimension of the safe array.
numElemsDim2	unsigned integer	Number of elements in the second dimension of the safe array.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_SafeArrayGetNumDims

```
HRESULT status = CA_SafeArrayGetNumDims (LPSAFEARRAY safeArray,
                                         unsigned int *numDims);
```

Purpose

Obtains the number of dimensions in a safe array.

Parameters

Input

Name	Type	Description
safeArray	LPSAFEARRAY	Safe array.

Output

Name	Type	Description
numDims	unsigned integer	Number of dimensions in the safe array.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_SafeArrayTo1DArray

```
HRESULT status = CA_SafeArrayTo1DArray (LPSAFEARRAY *safeArray,
                          unsigned int arrayType, void *array,
                          unsigned int *numElements);
```

Purpose

Converts a 1D safe array into a dynamically allocated C-style array.

Upon success, `CA_SafeArrayTo1DArray` frees the contents of the safe array and sets the safe array pointer to `NULL`.

Parameters

Input/Output

Name	Type	Description
safeArray	`LPSAFEARRAY`	1D safe array. Pass the address of the safe array pointer. `CA_SafeArrayTo1DArray` frees the safe array contents and sets the safe array pointer to `NULL`.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of the array `CA_SafeArrayTo1DArray` creates from the safe array.

Output

Name	Type	Description
array	depends on the value of **arrayType**	C-style array that `CA_SafeArrayTo1DArray` dynamically allocates. The type of the array must be the same as **arrayType**. Pass the address of an array pointer.
numElements	unsigned integer	Number of elements in **array**. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter must be the same as the type of the safe array except for the following cases:

- You can create a C-style array that contains char* elements from a BSTR safe array.
- You can create a C-style array that contains CAObjHandle elements from an LPDISPATCH safe array.

arrayType can be any of the data types in Table 11-2 except CAVT_EMPTY or CAVT_NULL. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. CA_SafeArrayTo1DArray ignores the CAVT_ARRAY modifier.

When you no longer need the C-style array, call CA_FreeMemory to discard it. If the C-style array contains elements of one of the data types in Table 11-10, use the corresponding function to free each element when you no longer need it.

Table 11-10. Data Types and Functions to Free Each Element for CA_SafeArrayTo1DArray

Data Type	Function to Free Each Element
char *	CA_FreeMemory
CAObjHandle	CA_DiscardObjHandle
BSTR	SysFreeString (a Windows SDK function)
LPUNKNOWN	Release (array[i]->lpVtbl->Release())
LPDISPATCH	Release (array[i]->lpVtbl->Release())
VARIANT	CA_VariantClear

Example

The following code shows you how to use `CA_SafeArrayTo1Darray`:

```
double * dblArray = NULL;
LPSAFEARRAY safeArray;
unsigned numElements;
int index;

/* Call an ActiveX Automation function that returns a safe array. */
.
.
.
/* Convert the safe array into a C-style array. */
CA_SafeArrayTo1DArray(&safeArray, CAVT_DOUBLE, &dblArray,
                      &numElements);

for (index = 0; index < numElements; index++)
   printf("%f", dblArray[index]);

/* Free the allocated array. */
CA_FreeMemory(dblArray);
```

CA_SafeArrayTo1DArrayBuf

```
HRESULT status = CA_SafeArrayTo1DArrayBuf (LPSAFEARRAY *safeArray,
                           unsigned int arrayType, void *arrayBuffer,
                           unsigned int bufferSize,
                           unsigned int *numElements);
```

Purpose

Converts a 1D safe array into a C-style array you pass as a buffer.

Upon success, `CA_SafeArrayTo1DArrayBuf` frees the contents of the safe array and sets the safe array pointer to `NULL`.

`CA_SafeArrayTo1DArrayBuf` returns an error if the buffer is not big enough to hold the array.

Parameters

Input/Output

Name	Type	Description
safeArray	LPSAFEARRAY	1D safe array. Pass the address of the safe array pointer. `CA_SafeArrayTo1DArrayBuf` frees the safe array contents and sets the safe array pointer to `NULL`.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of the array `CA_SafeArrayTo1DArrayBuf` creates from the safe array.
bufferSize	unsigned integer	Number of bytes in the **arrayBuffer** parameter.

Output

Name	Type	Description
arrayBuffer	depends on the value of **arrayType**	Buffer to receive the C-style array elements. The type of the array must be the same as **arrayType**.
numElements	unsigned integer	Number of elements `CA_SafeArrayTo1DArrayBuf` stores in **arrayBuffer**. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter must be the same as the type of the safe array except for the following cases:

- You can create a C-style array that contains `char*` elements from a `BSTR` safe array.
- You can create a C-style array that contains `CAObjHandle` elements from an `LPDISPATCH` safe array.

arrayType can be any of the data types in Table 11-2 except `CAVT_EMPTY` or `CAVT_NULL`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. `CA_SafeArrayTo1DArrayBuf` ignores the `CAVT_ARRAY` modifier.

If the C-style array contains elements of one of the data types in Table 11-11, use the corresponding function to free each element when you no longer need it.

Table 11-11. Data Types and Functions to Free Each CA_SafeArrayTo1DArrayBuf Element

Data Type	Function to Free Each Element
`char *`	`CA_FreeMemory`
`CAObjHandle`	`CA_DiscardObjHandle`
`BSTR`	`SysFreeString` (a Windows SDK function)
`LPUNKNOWN`	`Release(array[i]->lpVtbl->Release())`

Table 11-11. Data Types and Functions to Free Each CA_SafeArrayTo1DArrayBuf Element

Data Type	Function to Free Each Element
LPDISPATCH	Release(array[i]->lpVtbl->Release())
VARIANT	CA_VariantClear

Example

The following code shows you how to use CA_SafeArrayTo1DArrayBuf:

```
double dblArray[1024];
LPSAFEARRAY safeArray;
unsigned numElements;
int index;

/* Call an ActiveX Automation function that returns a safe array. */
.
.
.
/* Convert the safe array into a C-style array. */
CA_SafeArrayTo1DArrayBuf(&safeArray, CAVT_DOUBLE, dblArray,
                         sizeof(dblArray), &numElements);

for (index = 0; index < numElements; index++)
   printf("%f", dblArray[index]);
```

CA_SafeArrayTo2DArray

```
HRESULT status = CA_SafeArrayTo2DArray (LPSAFEARRAY *safeArray,
                          unsigned int arrayType, void *array,
                          unsigned int *numElemsDim1,
                          unsigned int *numElemsDim2);
```

Purpose

Converts a 2D safe array into a dynamically allocated C-style array.

Upon success, `CA_SafeArrayTo2DArray` frees the contents of the safe array and sets the safe array pointer to `NULL`.

Parameters

Input/Output

Name	Type	Description
safeArray	LPSAFEARRAY	2D safe array. Pass the address of the safe array pointer. `CA_SafeArrayTo2DArray` frees the safe array contents and sets the safe array pointer to `NULL`.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of the array `CA_SafeArrayTo2DArray` creates from the safe array.

Output

Name	Type	Description
array	depends on the value of **arrayType**	C-style array that `CA_SafeArrayTo2DArray` dynamically allocates. The type of the array must be the same as **arrayType**. Pass the address of an array pointer.
numElemsDim1	unsigned integer	Number of elements in the first dimension of **array**. You can pass `NULL` for this parameter.
numElemsDim2	unsigned integer	Number of elements in the second dimension of **array**. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter must be the same as the type of the safe array except for the following cases:

- You can create a C-style array that contains `char*` elements from a `BSTR` safe array.
- You can create a C-style array that contains `CAObjHandle` elements from an `LPDISPATCH` safe array.

arrayType can be any of the data types in Table 11-2 except `CAVT_EMPTY` or `CAVT_NULL`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. `CA_SafeArrayTo2DArray` ignores the `CAVT_ARRAY` modifier.

To access the elements of **array**, use the `CA_Get2DArrayElement` macro, which is declared in `cviauto.h`.

When you no longer need the C-style array, call `CA_FreeMemory` to discard it. If the C-style array contains elements of one of the data types in Table 11-12, use the corresponding function to free each element when you no longer need it.

Table 11-12. Data Types and Functions to Free Each Element for CA_SafeArrayTo2DArray

Data Type	Function to Free Each Element
`char *`	`CA_FreeMemory`
`CAObjHandle`	`CA_DiscardObjHandle`
`BSTR`	`SysFreeString` (a Windows SDK function)
`LPUNKNOWN`	`Release(array[i]->lpVtbl->Release())`
`LPDISPATCH`	`Release(array[i]->lpVtbl->Release())`
`VARIANT`	`CA_VariantClear`

Example

The following code shows you how to use `CA_SafeArrayTo2Darray`:

```
double *dblArray = NULL;
LPSAFEARRAY safeArray;
unsigned numElemsDim1, numElemsDim2;
int index1, index2;

/* Call an ActiveX Automation function that returns a safe array. */
.
.
.
/* Convert the safe array into a C-style array. */
CA_SafeArrayTo2DArray(&safeArray, CAVT_DOUBLE, &dblArray,
                      &numElemsDim1, &numElemsDim2);

for (index1 = 0; index1 < numElemsDim1; index1++)
   for (index2 = 0; index2 < numElemsDim2; index2++)
      {
      double d;
      d = CA_Get2DArrayElement(dblArray, numElemsDim1, numElemsDim2,
                               index1, index2, double);
      printf("%f", d);
      }

/* Free the allocated array. */
CA_FreeMemory(dblArray);
```

CA_SafeArrayTo2DArrayBuf

```
HRESULT status = CA_SafeArrayTo2DArrayBuf (LPSAFEARRAY *safeArray,
                           unsigned int arrayType, void *arrayBuffer,
                           unsigned int bufferSize,
                           unsigned int *numElemsDim1,
                           unsigned int *numElemsDim2);
```

Purpose

Converts a 2D safe array into a C-style array you pass in as a buffer.

Upon success, `CA_SafeArrayTo2DArrayBuf` frees the contents of the safe array and sets the safe array pointer to `NULL`.

`CA_SafeArrayTo2DArrayBuf` returns an error if the buffer is not big enough to hold the array.

Parameters

Input/Output

Name	Type	Description
safeArray	LPSAFEARRAY	2D safe array. Pass the address of the safe array pointer. `CA_SafeArrayTo2DArrayBuf` frees the safe array contents and sets the safe array pointer to `NULL`.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of the array `CA_SafeArrayTo2DArrayBuf` creates from the safe array.
bufferSize	unsigned integer	Number of bytes in the **arrayBuffer** parameter.

Output

Name	Type	Description
arrayBuffer	depends on the value of **arrayType**	Buffer to receive the C-style array elements. The type of the array must be the same as **arrayType**.
numElemsDim1	unsigned integer	Number of elements in the first dimension of **array**. You can pass `NULL` for this parameter.
numElemsDim2	unsigned integer	Number of elements in the second dimension of **array**. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter must be the same as the type of the safe array except for the following cases:

- You can create a C-style array that contains `char*` elements from a `BSTR` safe array.
- You can create a C-style array that contains `CAObjHandle` elements from an `LPDISPATCH` safe array.

arrayType can be any of the data types in Table 11-2 except `CAVT_EMPTY` or `CAVT_NULL`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. `CA_SafeArrayTo2DArrayBuf` ignores the `CAVT_ARRAY` modifier.

To access the elements of **array**, use the `CA_Get2DArrayElement` macro, which is declared in `cviauto.h`.

If the C-style array contains elements of one of the data types in Table 11-13, use the corresponding function to free each element when you no longer need it.

Table 11-13. Data Types and Functions to Free Each Element for CA_SafeArrayTo2DArrayBuf

Data Type	Function to Free Each Element
char*	`CA_FreeMemory`
`CAObjHandle`	`CA_DiscardObjHandle`
`BSTR`	`SysFreeString` (a Windows SDK function)
`LPUNKNOWN`	`Release(array[i]->lpVtbl->Release())`
`LPDISPATCH`	`Release(array[i]->lpVtbl->Release())`
`VARIANT`	`CA_VariantClear`

Example

The following code shows you how to use `CA_SafeArrayTo2DArrayBuf`:

```
double dblArray[1024];
LPSAFEARRAY safeArray;
unsigned numElemsDim1, numElemsDim2;
int index1, index2;

/* Call an ActiveX Automation function that returns a safe array. */
.
.
.
/* Convert the safe array into a C-style array. */
CA_SafeArrayTo2DArrayBuf(&safeArray, CAVT_DOUBLE, dblArray,
                         sizeof(dblArray), &numElemsDim1,
                         &numElemsDim2);

for (index1 = 0; index1 < numElemsDim1; index1++)
   for (index2 = 0; index2 < numElemsDim2; index2++)
      {
      double d;
      d = CA_Get2DArrayElement(dblArray, numElemsDim1, numElemsDim2,
                               index1, index2, double);
      printf("%f", d);
      }
```

CA_SetLocale

```
HRESULT status = CA_SetLocale (CAObjHandle objHandle, LCID locale);
```

Purpose

Sets the language the Automation server uses to interpret the arguments it receives as parameters to its functions.

If you do not call `CA_SetLocale`, the ActiveX Automation Library asks the Automation server to use `LANG_NEUTRAL`, which signifies the default language of the server.

Parameters

Input

Name	Type	Description
objHandle	`CAObjHandle`	ActiveX Object handle one of the object creation functions in this library or an Automation server method returns.
locale	`LCID`	ID that indicates the language the Automation server uses. Refer to the following *Parameter Discussion* section.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The locale can be any of the locales that the Automation server you are using supports. The following list shows the defined constant for locale ID:

```
LANG_NEUTRAL
LANG_AFRIKAANS
LANG_ALBANIAN
LANG_ARABIC
LANG_BASQUE
LANG_BELARUSIAN
LANG_BULGARIAN
LANG_CATALAN
LANG_CHINESE
```

```
LANG_CROATIAN
LANG_CZECH
LANG_DANISH
LANG_DUTCH
LANG_ENGLISH
LANG_ESTONIAN
LANG_FAEROESE
LANG_FARSI
LANG_FINNISH
LANG_FRENCH
LANG_GERMAN
LANG_GREEK
LANG_HEBREW
LANG_HUNGARIAN
LANG_ICELANDIC
LANG_INDONESIAN
LANG_ITALIAN
LANG_JAPANESE
LANG_KOREAN
LANG_LATVIAN
LANG_LITHUANIAN
LANG_NORWEGIAN
LANG_POLISH
LANG_PORTUGUESE
LANG_ROMANIAN
LANG_RUSSIAN
LANG_SERBIAN
LANG_SLOVAK
LANG_SLOVENIAN
LANG_SPANISH
LANG_SWEDISH
LANG_THAI
LANG_TURKISH
LANG_UKRAINIAN
LANG_VIETNAMESE
```

CA_VariantBool

```
VARIANT variant = CA_VariantBool (VBOOL boolValue);
```

Purpose

Converts a `VBOOL` value into a variant that contains the `VBOOL` value. Use `CA_VariantBool` to pass a `VBOOL` value as a variant parameter.

Parameter

Input

Name	Type	Description
boolValue	`VBOOL`	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **boolValue**.

CA_VariantBSTR

```
VARIANT variant = CA_VariantBSTR (BSTR BSTRValue);
```

Purpose

Converts a BSTR into a variant that contains a BSTR. Use CA_VariantBSTR to pass a BSTR value as a variant parameter.

Parameter

Input

Name	Type	Description
BSTRValue	BSTR	Value to store in the variant.

Return Value

Name	Type	Description
variant	VARIANT	Variant that contains the value in **BSTRValue**.

CA_VariantClear

```
HRESULT status = CA_VariantClear (VARIANT *variant);
```

Purpose

Frees the contents of a variant and marks the variant as empty.

Although you can call `CA_VariantClear` on a variant that contains a value of any type, `CA_VariantClear` frees resources only when the variant contains a string (`BSTR`), an automation object interface (`LPDISPATCH`), or an unknown interface (`LPUNKNOWN`).

`CA_VariantClear` always sets the variant type to `VT_EMPTY`.

Do not call `CA_VariantClear` on a variant that you have not initialized. You can initialize a variant using any of the `CA_VariantSet` functions.

Parameter

Input

Name	Type	Description
variant	`VARIANT`	Variant whose contents you want to clear.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantConvertToType

```
HRESULT status = CA_VariantConvertToType (VARIANT *variant,
                          unsigned int desiredType, void *convertedValue);
```

Purpose

Converts a value in a variant to a variable with a data type you specify. This can help you when you are uncertain about the data type a variant stores and when you need to work with a particular data type.

`CA_VariantConvertToType` converts all the fundamental types, such as numeric, string, `DATE`, `CURRENCY`, and so on. It converts from a pointer to a value by dereferencing the pointer. It cannot convert to a pointer type, and it cannot convert to or from array types unless the **desiredType** is exactly the same as the type in the variant.

Upon success, `CA_VariantConvertToType` frees the contents of the variant parameter, marks it as empty, and sets the type to `VT_EMPTY`.

Parameters

Input

Name	Type	Description
variant	`VARIANT`	Variant that contains the value to convert. `CA_VariantConvertToType` frees the variant contents and marks the variant as empty.
desiredType	unsigned integer	Type to convert the variant value to.

Output

Name	Type	Description
convertedValue	depends on the value of **desiredType**	Address of a variable large enough to hold the converted value.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

desiredType can be any of the fundamental data types in Table 11-2 except `CAVT_EMPTY` or `CAVT_NULL`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. You can use the `CAVT_ARRAY` modifier.

Note ***If you use the*** `CAVT_ARRAY` ***modifier,*** `CA_VariantConvertToType` ***returns a safe array, not a C-style array.***

If the **desiredType** is one of the data types in Table 11-14, you must call the corresponding function to free the **convertedValue** when you no longer need it.

Table 11-14. Data Types and Functions to Free the Converted Value

Data Type	Function to Free Converted Value
`char *`	`CA_FreeMemory`
`CAObjHandle`	`CA_DiscardObjHandle`
`BSTR`	`SysFreeString` (a Windows SDK function)
`LPUNKNOWN`	`Release(convertedValue->lpVtbl.Release())`
`LPDISPATCH`	`Release(convertedValue->lpVtbl.Release())`
any type | `CAVT_ARRAY`	`CA_SafeArrayDestroy`

CA_VariantCopy

```
HRESULT status = CA_VariantCopy (VARIANT *sourceVariant,
                                 VARIANT *destinationVariant);
```

Purpose

Copies the contents of one variant to another variant. `CA_VariantCopy` makes a deep copy of the source variant by duplicating any allocated data that it contains.

Parameters

Input

Name	Type	Description
sourceVariant	pointer to `VARIANT`	Pointer to the variant that contains the value to copy.

Output

Name	Type	Description
destinationVariant	`VARIANT`	Variant into which `CA_VariantCopy` copies the **sourceVariant** contents.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\3sdk\winerror.h` for Windows SDK error codes.

CA_VariantCurrency

```
VARIANT variant = CA_VariantCurrency (CURRENCY currencyValue);
```

Purpose

Converts a CURRENCY value into a variant that contains the CURRENCY value. Use CA_VariantCurrency to pass a CURRENCY value as a variant parameter.

Parameter

Input

Name	Type	Description
currencyValue	CURRENCY	Value to store in the variant.

Return Value

Name	Type	Description
variant	VARIANT	Variant that contains the value in **currencyValue**.

CA_VariantDate

```
VARIANT variant = CA_VariantDate (DATE dateValue);
```

Purpose

Converts a DATE value into a variant that contains the DATE value. Use CA_VariantDate to pass a DATE value as a variant parameter.

Parameter

Input

Name	Type	Description
dateValue	DATE	Value to store in the variant.

Return Value

Name	Type	Description
variant	VARIANT	Variant that contains the value in **dateValue**.

CA_VariantDispatch

```
VARIANT variant = CA_VariantDispatch (LPDISPATCH dispatchValue);
```

Purpose

Converts a `LPDISPATCH` value into a variant that contains the `LPDISPATCH` value. Use `CA_VariantDispatch` to pass a `LPDISPATCH` value as a variant parameter.

Parameter

Input

Name	Type	Description
dispatchValue	`LPDISPATCH`	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **dispatchValue**.

CA_VariantDouble

```
VARIANT variant = CA_VariantDouble (double doubleValue);
```

Purpose

Converts a double-precision value into a variant that contains the double-precision value. Use `CA_VariantDouble` to pass a double-precision value as a variant parameter.

Parameter

Input

Name	Type	Description
doubleValue	double-precision	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **doubleValue**.

CA_VariantEmpty

```
VARIANT variant = CA_VariantEmpty (void);
```

Purpose

Returns a variant in which the value of the type field is `VT_EMPTY`. Use `CA_VariantEmpty` to pass an empty variant as a parameter.

Parameters

None.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant in which the value of the type field is `VT_EMPTY`.

CA_VariantError

```
VARIANT variant = CA_VariantError (SCODE errorValue);
```

Purpose

Converts an SCODE value into a variant that contains the SCODE value. SCODE is the data type for an error value. Use CA_VariantError to pass an SCODE value as a variant parameter.

Parameter

Input

Name	Type	Description
errorValue	SCODE	Value to store in the variant.

Return Value

Name	Type	Description
variant	VARIANT	Variant that contains the value in **errorValue**.

CA_VariantFloat

```
VARIANT variant = CA_VariantFloat (float floatValue);
```

Purpose

Converts a single-precision, floating-point value into a variant that contains the single-precision value. Use `CA_VariantFloat` to pass a single-precision value as a variant parameter.

Parameter

Input

Name	Type	Description
floatValue	single-precision	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **floatValue**.

CA_VariantGet1DArray

```
HRESULT status = CA_VariantGet1DArray (VARIANT *variant,
                          unsigned int arrayType, void *array,
                          unsigned int *numElements);
```

Purpose

Converts a 1D safe array in a variant parameter into a dynamically allocated C-style array.

Upon success, `CA_VariantGet1DArray` frees the contents of the variant parameter and marks it as empty.

Parameters

Input/Output

Name	Type	Description
variant	`VARIANT`	Pointer to a variant that contains a 1D safe array. `CA_VariantGet1DArray` frees the contents of the variant contents and marks it as empty.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of the array `CA_VariantGet1DArray` creates from the safe array.

Output

Name	Type	Description
array	depends on the value of **arrayType**	C-style array that `CA_VariantGet1DArray` dynamically allocates. The type of the array must be the same as **arrayType**. Pass the address of an array pointer.
numElements	unsigned integer	Number of elements in **array**. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter must be the same as the type of the safe array except for the following cases:

- You can create a C-style array that contains char* elements from a BSTR safe array.
- You can create a C-style array that contains CAObjHandle elements from an LPDISPATCH safe array.

arrayType can be any of the data types in Table 11-2 except CAVT_EMPTY or CAVT_NULL. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. CA_VariantGet1DArray ignores the CAVT_ARRAY modifier.

If you do not know the type of the array, you can call CA_VariantGetType and pass its return value as the **arrayType**.

When you no longer need the C-style array, call CA_FreeMemory to discard it. If the C-style array contains elements of one of the data types in Table 11-15, use the corresponding function to free each element when you no longer need it.

Table 11-15. Data Types and Functions to Free Each Element for CA_VariantGet1DArray

Data Type	Function to Free Each Element
char *	CA_FreeMemory
CAObjHandle	CA_DiscardObjHandle
BSTR	SysFreeString (a Windows SDK function)
LPUNKNOWN	Release (array[i]->lpVtbl->Release())
LPDISPATCH	Release (array[i]->lpVtbl->Release())
VARIANT	CA_VariantClear

Example

The following code shows you how to use `CA_VariantGet1Darray`:

```
double *dblArray = NULL;
VARIANT variant;
unsigned numElements;
int index;

/* Call an ActiveX Automation function that returns a safe array in a
Variant. */
.
.
.
/* Convert the safe array the variant contains into a C-style array. */
CA_VariantGet1DArray(&variant, CAVT_DOUBLE, &dblArray, &numElements);

for (index = 0; index < numElements; index++)
   printf("%f", dblArray[index]);

/* Free the allocated array. */
CA_FreeMemory(dblArray);
```

CA_VariantGet1DArrayBuf

```
HRESULT status = CA_VariantGet1DArrayBuf (VARIANT *variant,
                          unsigned int arrayType, void *arrayBuffer,
                          unsigned int bufferSize,
                          unsigned int *numElements);
```

Purpose

Converts a 1D safe array in a variant parameter into a C-style array you pass as a buffer.

On success, `CA_VariantGet1DArrayBuf` releases the contents of the variant parameter and marks it as empty.

`CA_VariantGet1DArrayBuf` returns an error if the buffer is not big enough to hold the array.

Parameters

Input/Output

Name	Type	Description
variant	`VARIANT`	Pointer to a variant that contains a 1D safe array. `CA_VariantGet1DArrayBuf` frees the contents of the variant contents and marks it as empty.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of the array `CA_VariantGet1DArrayBuf` creates from the safe array.
bufferSize	unsigned integer	Number of bytes in the **arrayBuffer** parameter.

Output

Name	Type	Description
arrayBuffer	depends on the value of **arrayType**	Buffer to receive the C-style array elements. The type of the array must be the same as **arrayType**.
numElements	unsigned integer	Number of elements `CA_VariantGet1DArrayBuf` stores in **arrayBuffer**. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter must be the same as the type of the safe array except for the following cases:

- You can create a C-style array that contains `char*` elements from a `BSTR` safe array.
- You can create a C-style array that contains `CAObjHandle` elements from an `LPDISPATCH` safe array.

arrayType can be any of the data types in Table 11-2 except `CAVT_EMPTY` or `CAVT_NULL`.Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. `CA_VariantGet1DArrayBuf` ignores the `CAVT_ARRAY` modifier.

If you do not know the type of the array, you can call `CA_VariantGetType` and pass its return value as the **arrayType**.

If the C-style array contains elements of one of the data types in Table 11-16, use the corresponding function to free each element when you no longer need it.

Table 11-16. Data Types and Functions to Free Each Element for CA_VariantGet1DArrayBuf

Data Type	Function to Free Each Element
`char *`	`CA_FreeMemory`
`CAObjHandle`	`CA_DiscardObjHandle`
`BSTR`	`SysFreeString` (a Windows SDK function)
`LPUNKNOWN`	`Release(array[i]->lpVtbl->Release())`
`LPDISPATCH`	`Release(array[i]->lpVtbl->Release())`
`VARIANT`	`CA_VariantClear`

Example

The following code shows you how to use `CA_VariantGet1DArrayBuf`:

```
double dblArray[1024];
VARIANT variant;
unsigned numElements;
int index;

/* Call an ActiveX Automation function that returns a safe array in a
Variant. */
.
.
.
/* Convert the safe array the variant contains into a C-style array. */
CA_VariantGet1DArrayBuf(&variant, CAVT_DOUBLE, dblArray,
                        sizeof(dblArray), &numElements);
for (index = 0; index < numElements; index++)
   printf("%f", dblArray[index]);
```

CA_VariantGet1DArraySize

```
HRESULT status = CA_VariantGet1DArraySize (VARIANT *variant,
                                           unsigned int *numElements);
```

Purpose

Obtains the number of elements in a 1D safe array in the variant you specify.

Parameters

Input

Name	Type	Description
variant	`VARIANT`	Pointer to a variant that contains a 1D safe array.

Output

Name	Type	Description
numElements	unsigned integer	Number of elements in the safe array.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGet2DArray

```
HRESULT status = CA_VariantGet2DArray (VARIANT *variant,
                          unsigned int arrayType, void *array,
                          unsigned int *numElemsDim1,
                          unsigned int *numElemsDim2);
```

Purpose

Converts a 2D safe array in a variant parameter into a dynamically allocated C-style array.

Upon success, `CA_VariantGet2DArray` frees the contents of the variant parameter and marks it as empty.

Parameters

Input/Output

Name	Type	Description
variant	`VARIANT`	Pointer to a variant that contains a 2D safe array. `CA_VariantGet2DArray` frees the contents of the variant contents and marks it as empty.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of the array `CA_VariantGet2DArray` creates from the safe array.

Output

Name	Type	Description
array	depends on the value of **arrayType**	C-style array that `CA_VariantGet2DArray` dynamically allocates. The type of the array must be the same as **arrayType**. Pass the address of an array pointer.
numElemsDim1	unsigned integer	Number of elements in the first dimension of **array**. You can pass `NULL` for this parameter.
numElemsDim2	unsigned integer	Number of elements in the second dimension of **array**. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter must be the same as the type of the safe array except for the following cases:

- You can create a C-style array that contains `char*` elements from a `BSTR` safe array.
- You can create a C-style array that contains `CAObjHandle` elements from an `LPDISPATCH` safe array.

arrayType can be any of the data types in Table 11-2 except `CAVT_EMPTY` or `CAVT_NULL`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. `CA_VariantGet2Darray` ignores the `CAVT_ARRAY` modifier.

If you do not know the type of the array, you can call `CA_VariantGetType` and pass its return value as the **arrayType**.

To access the elements of **array**, use the `CA_Get2DArrayElement` macro, which is declared in `cviauto.h`.

When you no longer need the C-style array, call `CA_FreeMemory` to discard it. If the C-style array contains elements of one of the data types in Table 11-17, use the corresponding function to free each element when you no longer need it.

Table 11-17. Data Types and Functions to Free Each Element for CA_VariantGet2DArray

Data Type	Function to Free Each Element
`char *`	`CA_FreeMemory`
`CAObjHandle`	`CA_DiscardObjHandle`
`BSTR`	`SysFreeString` (a Windows SDK function)
`LPUNKNOWN`	`Release(array[i]->lpVtbl->Release())`
`LPDISPATCH`	`Release(array[i]->lpVtbl->Release())`
`VARIANT`	`CA_VariantClear`

Example

The following code shows you how to use `CA_VariantGet2Darray`:

```
double *dblArray = NULL;
VARIANT variant;
unsigned numElemsDim1, numElemsDim2;
int index1, index2;

/* Call an ActiveX Automation function that returns a safe array in a
Variant. */
.
.
.
/* Convert the safe array the variant contains into a C-style array. */
CA_VariantGet2DArray(&variant, CAVT_DOUBLE, &dblArray, &numElemsDim1,
                     &numElemsDim2);

for (index1 = 0; index1 < numElemsDim1; index1++)
   for (index2 = 0; index2 < numElemsDim2; index2++)
      {
      double d;
      d = CA_Get2DArrayElement(dblArray, numElemsDim1, numElemsDim2,
                               index1, index2, double);
      printf("%f", d);
      }

/* Free the allocated array. */
CA_FreeMemory(dblArray);
```

CA_VariantGet2DArrayBuf

```
HRESULT status = CA_VariantGet2DArrayBuf (VARIANT *variant,
                            unsigned int arrayType, void *arrayBuffer,
                            unsigned int bufferSize,
                            unsigned int *numElemsDim1,
                            unsigned int *numElemsDim2);
```

Purpose

Converts a 2D safe array in a variant parameter into a C-style array you pass as a buffer.

On success, `CA_VariantGet2DArrayBuf` releases the contents of the variant parameter and marks it as empty.

`CA_VariantGet2DArrayBuf` returns an error if the buffer is not big enough to hold the array.

Parameters

Input/Output

Name	Type	Description
variant	`VARIANT`	Pointer to a variant that contains a 2D safe array. `CA_VariantGet2DArrayBuf` frees the contents of the variant contents and marks it as empty.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of the array `CA_VariantGet2DArrayBuf` creates from the safe array.
bufferSize	unsigned integer	Number of bytes in the **arrayBuffer** parameter.

Output

Name	Type	Description
arrayBuffer	depends on the value of **arrayType**	Buffer to receive the C-style array elements. The type of the array must be the same as **arrayType**.
numElemsDim1	unsigned integer	Number of elements in the first dimension of **array**. You can pass `NULL` for this parameter.
numElemsDim2	unsigned integer	Number of elements in the second dimension of **array**. You can pass `NULL` for this parameter.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter must be the same as the type of the safe array except for the following cases:

- You can create a C-style array that contains `char*` elements from a `BSTR` safe array.
- You can create a C-style array that contains `CAObjHandle` elements from an `LPDISPATCH` safe array.

arrayType can be any of the data types in Table 11-2 except `CAVT_EMPTY` or `CAVT_NULL`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter. `CA_VariantGet2DArrayBuf` ignores the `CAVT_ARRAY` modifier.

If you do not know the type of the array, you can call `CA_VariantGetType` and pass its return value as the **arrayType**.

To access the elements of **array**, use the `CA_Get2DArrayElement` macro, which is declared in `cviauto.h`.

If the C-style array contains elements of one of the data types in Table 11-18, use the corresponding function to free each element when you no longer need it.

Table 11-18. Data Types and Functions to Free Each Element for CA_VariantGet2DArrayBuf

Data Type	Function to Free Each Element
`char *`	`CA_FreeMemory`
`CAObjHandle`	`CA_DiscardObjHandle`
`BSTR`	`SysFreeString` (a Windows SDK function)
`LPUNKNOWN`	`Release(array[i]->lpVtbl->Release())`
`LPDISPATCH`	`Release(array[i]->lpVtbl->Release())`
`VARIANT`	`CA_VariantClear`

Example

The following code shows you how to use `CA_VariantGet2DArrayBuf`:

```
double dblArray[1024];
VARIANT variant;
unsigned numElemsDim1, numElemsDim2;
int index1, index2;

/* Call an ActiveX Automation function that returns a safe array in a
Variant. */
.
.
.
/* Convert the safe array the variant contains into a C-style array. */
CA_VariantGet2DArrayBuf(&variant, CAVT_DOUBLE, dblArray,
                        sizeof(dblArray), &numElemsDim1,
&numElemsDim2);

for (index1 = 0; index1 < numElemsDim1; index1++)
   for (index2 = 0; index2 < numElemsDim2; index2++)
      {
      double d;
      d = CA_Get2DArrayElement(dblArray, numElemsDim1, numElemsDim2,
                               index1, index2, double);
      printf("%f", d);
      }
```

CA_VariantGet2DArraySize

```
HRESULT status = CA_VariantGet2DArraySize (VARIANT *variant,
                          unsigned int *numElemsDim1,
                          unsigned int *numElemsDim2);
```

Purpose

Obtains the number of elements in a 2D safe array in the variant parameter you specify.

Parameters

Input

Name	Type	Description
variant	`VARIANT`	Pointer to a variant that contains a 2D safe array.

Output

Name	Type	Description
numElemsDim1	unsigned integer	Number of elements in the first dimension of the safe array.
numElemsDim2	unsigned integer	Number of elements in the second dimension of the safe array.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetArrayNumDims

```
HRESULT status = CA_VariantGetArrayNumDims (VARIANT *variant,
                                            unsigned int *numDims);
```

Purpose

Obtains the number of dimensions in a safe array in the variant you specify.

Parameters

Input

Name	Type	Description
variant	`VARIANT`	Pointer to a variant that contains a safe array.

Output

Name	Type	Description
numDims	unsigned integer	Number of dimensions in the safe array.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetBool

```
HRESULT status = CA_VariantGetBool (VARIANT *variant, VBOOL *boolValue);
```

Purpose

Copies the value in a variant into a `VBOOL` variable.

`CA_VariantGetBool` returns an error if the variant does not contain a `VBOOL` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a `VBOOL` value.

Output

Name	Type	Description
boolValue	`VBOOL`	`VBOOL` value `CA_VariantGetBool` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetBoolPtr

```
HRESULT status = CA_VariantGetBoolPtr (VARIANT *variant,
                                       VBOOL **boolValuePtr);
```

Purpose

Copies the value in a variant into a `VBOOL` pointer variable.

`CA_VariantGetBoolPtr` returns an error if the variant does not contain a pointer to a `VBOOL` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a `VBOOL` pointer.

Output

Name	Type	Description
boolValuePtr	pointer to `VBOOL`	`VBOOL` pointer `CA_VariantGetBoolPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetBSTR

```
HRESULT status = CA_VariantGetBSTR (VARIANT *variant, BSTR *BSTRValue);
```

Purpose

Copies the value in a variant into a BSTR variable.

On success, CA_VariantGetBSTR marks the variant as empty.

CA_VariantGetBSTR returns an error if the variant does not contain a BSTR value.

Parameters

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to a variant that contains a BSTR value. CA_VariantGetBSTR marks the variant as empty on success.

Output

Name	Type	Description
BSTRValue	BSTR	BSTR value CA_VariantGetBSTR copies from the variant.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When you no longer need the BSTR, call the Windows SDK function SysFreeString to free it.

CA_VariantGetBSTRPtr

```
HRESULT status = CA_VariantGetBSTRPtr (VARIANT *variant,
                                       BSTR **BSTRValuePtr);
```

Purpose

Copies the value in a variant into a BSTR pointer variable.

CA_VariantGetBSTRPtr returns an error if the variant does not contain a pointer to a BSTR value.

Parameters

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to a variant that contains a BSTR pointer.

Output

Name	Type	Description
BSTRValuePtr	pointer to BSTR	BSTR pointer CA_VariantGetBSTRPtr copies from the variant.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantGetCString

```
HRESULT status = CA_VariantGetCString (VARIANT *variant, char **cString);
```

Purpose

Converts the BSTR string in the variant you specify to a dynamically allocated C-style string.

On success, CA_VariantGetCString releases the contents of the variant and marks it as empty.

CA_VariantGetCString returns an error if the variant does not contain a BSTR value.

Parameters

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to a variant that contains a BSTR value. CA_VariantGetCString marks the variant as empty on success.

Output

Name	Type	Description
cString	string	Dynamically allocated C-style string CA_VariantGetCString converts from the BSTR in the variant.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When you no longer need the C-style string, call CA_FreeMemory to free it.

CA_VariantGetCStringBuf

```
HRESULT status = CA_VariantGetCStringBuf (VARIANT *variant,
                          char buffer[], unsigned long bufferSize);
```

Purpose

Converts the BSTR string in the variant you specify to a C-style string and copies the string into a buffer parameter.

If **buffer** is not large enough to hold the string, CA_VariantGetCStringBuf copies (**bufferSize** –1) characters into the buffer, followed by an ASCII NUL byte.

On success, CA_VariantGetCStringBuf releases the contents of the variant and marks it as empty.

CA_VariantGetCStringBuf returns an error if the variant does not contain a BSTR value.

Parameters

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to a variant that contains a BSTR value. CA_VariantGetCStringBuf marks the variant as empty on success.
bufferSize	unsigned integer	Number of bytes in **buffer**.

Output

Name	Type	Description
buffer	character array	Buffer into which CA_VariantGetCStringBuf copies the C-style string it converts from the BSTR in the variant.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantGetCStringLen

```
HRESULT status = CA_VariantGetCStringLen (VARIANT *variant,
                          unsigned long *len);
```

Purpose

Obtains the length of the C string you can create by calling `CA_VariantGetCString` to convert the `BSTR` string in the variant you specify. The length does not include the ASCII NUL byte.

`CA_VariantGetCStringLen` returns an error if the variant does not contain a `BSTR` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a `BSTR` value.

Output

Name	Type	Description
len	string	Length of C-style string that `CA_VariantGetCString` can convert from the BSTR in the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetCurrency

```
HRESULT status = CA_VariantGetCurrency (VARIANT *variant,
                                        CURRENCY *currencyValue);
```

Purpose

Copies the value in a variant into a CURRENCY variable.

CA_VariantGetCurrency returns an error if the variant does not contain a CURRENCY value.

Parameters

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to a variant that contains a CURRENCY value.

Output

Name	Type	Description
currencyValue	CURRENCY	CURRENCY value CA_VariantGetCurrency copies from the variant.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantGetCurrencyPtr

```
HRESULT status = CA_VariantGetCurrencyPtr (VARIANT *variant,
                          CURRENCY **currencyValuePtr);
```

Purpose

Copies the value in a variant into a `CURRENCY` pointer variable.

`CA_VariantGetCurrencyPtr` returns an error if the variant does not contain a pointer to a `CURRENCY` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a `CURRENCY` pointer.

Output

Name	Type	Description
currencyValuePtr	pointer to `CURRENCY`	`CURRENCY` pointer `CA_VariantGetCurrencyPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetDate

```
HRESULT status = CA_VariantGetDate (VARIANT *variant, DATE *dateValue);
```

Purpose

Copies the value in a variant into a DATE variable.

CA_VariantGetDate returns an error if the variant does not contain a DATE value.

Parameters

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to a variant that contains a DATE value.

Output

Name	Type	Description
dateValue	DATE	DATE value CA_VariantGetDate copies from the variant.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantGetDatePtr

```
HRESULT status = CA_VariantGetDatePtr (VARIANT *variant,
                                       DATE **dateValuePtr);
```

Purpose

Copies the value in a variant into a `DATE` pointer variable.

`CA_VariantGetDatePtr` returns an error if the variant does not contain a pointer to a `DATE` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a `DATE` pointer.

Output

Name	Type	Description
dateValuePtr	pointer to `DATE`	`DATE` pointer `CA_VariantGetDatePtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetDispatch

```
HRESULT status = CA_VariantGetDispatch (VARIANT *variant,
                          LPDISPATCH *dispatchValue);
```

Purpose

Copies the value in a variant into an `LPDISPATCH` variable. An `LPDISPATCH` value is a dispatch pointer for an ActiveX Automation object interface.

On success, `CA_VariantGetDispatch` marks the variant parameter as empty.

`CA_VariantGetDispatch` returns an error if the variant does not contain an `LPDISPATCH` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an `LPDISPATCH` value. `CA_VariantGetDispatch` marks the variant as empty on success.

Output

Name	Type	Description
dispatchValue	`LPDISPATCH`	`LPDISPATCH` value `CA_VariantGetDispatch` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

When you no longer need the `LPDISPATCH`, free it by calling its `Release` function, as in the following:

```
lpDispatch->lpVtbl->Release();
```

CA_VariantGetDispatchPtr

```
HRESULT status = CA_VariantGetDispatchPtr (VARIANT *variant,
                          LPDISPATCH **dispatchValuePtr);
```

Purpose

Copies the value in a variant into a variable that is a pointer to the `LPDISPATCH` type. An `LPDISPATCH` value is a dispatch pointer for an ActiveX Automation object interface.

`CA_VariantGetDispatchPtr` returns an error if the variant does not contain a pointer to an `LPDISPATCH` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an `LPDISPATCH` pointer.

Output

Name	Type	Description
dispatchValuePtr	pointer to `LPDISPATCH`	`LPDISPATCH` pointer `CA_VariantGetDispatchPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetDouble

```
HRESULT status = CA_VariantGetDouble (VARIANT *variant,
                                      double *dblValue);
```

Purpose

Copies the value in a variant into a double-precision variable.

`CA_VariantGetDouble` returns an error if the variant does not contain a double-precision value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a double-precision value.

Output

Name	Type	Description
dblValue	double-precision	Double-precision value `CA_VariantGetDouble` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetDoublePtr

```
HRESULT status = CA_VariantGetDoublePtr (VARIANT *variant,
                          double **doubleValuePtr);
```

Purpose

Copies the value in a variant into a double-precision pointer variable.

`CA_VariantGetDoublePtr` returns an error if the variant does not contain a pointer to a double-precision value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a double-precision pointer.

Output

Name	Type	Description
doubleValuePtr	pointer to double-precision	Double-precision pointer `CA_VariantGetDoublePtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetError

```
HRESULT status = CA_VariantGetError (VARIANT *variant,
                                     SCODE *errorValue);
```

Purpose

Copies the value in a variant into an `SCODE` variable. `SCODE` is the data type for an error value.

`CA_VariantGetError` returns an error if the variant does not contain an `SCODE` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an `SCODE` value.

Output

Name	Type	Description
errorValue	`SCODE`	`SCODE` value `CA_VariantGetError` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetErrorPtr

```
HRESULT status = CA_VariantGetErrorPtr (VARIANT *variant,
                                        SCODE **errorValuePtr);
```

Purpose

Copies the value in a variant into an SCODE pointer variable. SCODE is the data type for an error value.

CA_VariantGetErrorPtr returns an error if the variant does not contain a pointer to an SCODE value.

Parameters

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to a variant that contains an SCODE pointer.

Output

Name	Type	Description
errorValuePtr	pointer to SCODE	SCODE pointer CA_VariantGetErrorPtr copies from the variant.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantGetFloat

```
HRESULT status = CA_VariantGetFloat (VARIANT *variant,
                                     float *floatValue);
```

Purpose

Copies the value in a variant into a single-precision, floating-point variable.

`CA_VariantGetFloat` returns an error if the variant does not contain a single-precision value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a single-precision value.

Output

Name	Type	Description
floatValue	single-precision	Single-precision value `CA_VariantGetFloat` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetFloatPtr

```
HRESULT status = CA_VariantGetFloatPtr (VARIANT *variant,
                                        float **floatValuePtr);
```

Purpose

Copies the value in a variant into a single-precision, floating-point pointer variable.

`CA_VariantGetFloatPtr` returns an error if the variant does not contain a pointer to a single-precision value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a single-precision pointer.

Output

Name	Type	Description
floatValuePtr	pointer to single-precision	Single-precision pointer `CA_VariantGetFloatPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetInt

```
HRESULT status = CA_VariantGetInt (VARIANT *variant, int *intValue);
```

Purpose

Copies the value in a variant into an integer variable.

`CA_VariantGetInt` returns an error if the variant does not contain an integer value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an integer value.

Output

Name	Type	Description
intValue	integer	Integer value `CA_VariantGetInt` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetIntPtr

```
HRESULT status = CA_VariantGetIntPtr (VARIANT *variant,
                                      int **intValuePtr);
```

Purpose

Copies the value in a variant into an integer pointer variable.

`CA_VariantGetIntPtr` returns an error if the variant does not contain a pointer to an integer value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an integer pointer.

Output

Name	Type	Description
intValuePtr	pointer to integer	Integer pointer `CA_VariantGetIntPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetIUnknown

```
HRESULT status = CA_VariantGetIUnknown (VARIANT *variant,
                            LPUNKNOWN *IUnknownValue);
```

Purpose

Copies the value in a variant into a LPUNKNOWN variable. An LPUNKNOWN value is a pointer to an unknown interface.

On success, CA_VariantGetIUnknown marks the variant parameter as empty.

CA_VariantGetIUnknown returns an error if the variant does not contain an LPUNKNOWN value.

Parameters

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to a variant that contains an LPUNKNOWN value. CA_VariantGetIUnknown marks the variant as empty on success.

Output

Name	Type	Description
IUnknownValue	LPUNKNOWN	LPUNKNOWN value CA_VariantGetIUnknown copies from the variant.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

Parameter Discussion

When you no longer need the LPUNKNOWN, free it by calling its Release function, as in the following:

```
lpUnknown->lpVtbl->Release();
```

CA_VariantGetIUnknownPtr

```
HRESULT status = CA_VariantGetIUnknownPtr (VARIANT *variant,
                                           LPUNKNOWN **IUnknownValPtr);
```

Purpose

Copies the value in a variant into a variable that is a pointer to the `LPUNKNOWN` type. An `LPUNKNOWN` value is a pointer to an unknown interface.

`CA_VariantGetIUnknownPtr` returns an error if the variant does not contain a pointer to an `LPUNKNOWN` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an `LPUNKNOWN` pointer.

Output

Name	Type	Description
IUnknownValPtr	pointer to `LPUNKNOWN`	`LPUNKNOWN` pointer `CA_VariantGetIUnknownPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetLong

```
HRESULT status = CA_VariantGetLong (VARIANT *variant, long *longValue);
```

Purpose

Copies the value in a variant into a long integer variable.

`CA_VariantGetLong` returns an error if the variant does not contain a long integer value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a long integer value.

Output

Name	Type	Description
longValue	long integer	Long integer value `CA_VariantGetLong` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetLongPtr

```
HRESULT status = CA_VariantGetLongPtr (VARIANT *variant,
                                       long **longValuePtr);
```

Purpose

Copies the value in a variant into a long integer pointer variable.

`CA_VariantGetLongPtr` returns an error if the variant does not contain a pointer to a long integer value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a long integer pointer.

Output

Name	Type	Description
longValuePtr	pointer to long integer	long integer pointer `CA_VariantGetLongPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetObjHandle

```
HRESULT status = CA_VariantGetObjHandle (VARIANT *variant,
                                         CAObjHandle *objHandle);
```

Purpose

Converts the `LPDISPATCH` value in a variant to a `CAObjHandle`.

On success, `CA_VariantGetObjHandle` marks the variant parameter as empty.

`CA_VariantGetObjHandle` returns an error if the variant does not contain an `LDISPATCH` value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an `LPDISPATCH` value. `CA_VariantGetObjHandle` marks the variant as empty on success.

Output

Name	Type	Description
objHandle	`CAObjHandle`	Object handle `CA_VariantGetObjHandle` converts from the `LPDISPATCH` value in the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

When you no longer need **objHandle**, call `CA_DiscardObjHandle` to free it.

CA_VariantGetSafeArray

```
HRESULT status = CA_VariantGetSafeArray (VARIANT *variant,
                          unsigned int arrayType,
                          LPSAFEARRAY *safeArray);
```

Purpose

Copies the safe array in a variant into a safe array variable.

On success, `CA_VariantGetSafeArray` marks the variant parameter as empty.

`CA_VariantGetSafeArray` returns an error if the variant does not contain a safe array.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a safe array. `CA_VariantGetSafeArray` marks the variant as empty on success.
arrayType	unsigned integer	Type of the safe array.

Output

Name	Type	Description
safeArray	`LPSAFEARRAY`	Safe array `CA_VariantGetSafeArray` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter can have any of the values in Table 11-2 except for `CAVT_EMPTY`, `CAVT_NULL`, and `CAVT_OBJHANDLE`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

If you do not know the type of the safe array, call `CA_VariantGetType` and pass its return value as the **arrayType**. `CA_VariantGetSafeArray` ignores the `CAVT_ARRAY` modifier.

CA_VariantGetSafeArrayPtr

```
HRESULT status = CA_VariantGetSafeArrayPtr (VARIANT *variant,
                            unsigned int arrayType,
                            LPSAFEARRAY **safeArrayPtr);
```

Purpose

Copies the value in a variant into a safe array pointer variable.

`CA_VariantGetSafeArrayPtr` returns an error if the variant does not contain a pointer to a safe array.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a pointer to a safe array.
arrayType	unsigned integer	Type of the safe array.

Output

Name	Type	Description
safeArrayPtr	pointer to `LPSAFEARRAY`	Safe array pointer `CA_VariantGetSafeArrayPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter can have any of the values in Table 11-2 except for `CAVT_EMPTY`, `CAVT_NULL`, and `CAVT_OBJHANDLE`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

If you do not know the type of the safe array, call `CA_VariantGetType` and pass its return value as the **arrayType**. `CA_VariantGetSafeArrayPtr` ignores the `CAVT_ARRAY` modifier.

CA_VariantGetShort

```
HRESULT status = CA_VariantGetShort (VARIANT *variant,
                                     short *shortValue);
```

Purpose

Copies the value in a variant into a short integer variable.

`CA_VariantGetShort` returns an error if the variant does not contain a short integer value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a short integer value.

Output

Name	Type	Description
shortValue	short integer	Short integer value `CA_VariantGetShort` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetShortPtr

```
HRESULT status = CA_VariantGetShortPtr (VARIANT *variant,
                                        short **shortValuePtr);
```

Purpose

Copies the value in a variant into a short integer pointer variable.

`CA_VariantGetShortPtr` returns an error if the variant does not contain a pointer to a short integer value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a short integer pointer.

Output

Name	Type	Description
shortValuePtr	pointer to short integer	Short integer pointer `CA_VariantGetShortPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetType

```
unsigned int type = CA_VariantGetType (VARIANT *variant);
```

Purpose

Returns the type of value the variant you specify contains.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
type	unsigned integer	0 if the parameter is invalid. Otherwise, one of the data types listed in Table 11-2 in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter, possibly bitwise OR'ed with one or more of the data type modifiers listed in Table 11-3 in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

CA_VariantGetUChar

```
HRESULT status = CA_VariantGetUChar (VARIANT *variant,
                                     unsigned char *uCharValue);
```

Purpose

Copies the value in a variant into an unsigned character variable.

`CA_VariantGetUChar` returns an error if the variant does not contain an unsigned character value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an unsigned character value.

Output

Name	Type	Description
uCharValue	unsigned character	unsigned character value `CA_VariantGetUChar` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetUCharPtr

```
HRESULT status = CA_VariantGetUCharPtr (VARIANT *variant,
                          unsigned char **uCharValuePtr);
```

Purpose

Copies the value in a variant into an unsigned character pointer variable.

`CA_VariantGetUCharPtr` returns an error if the variant does not contain a pointer to an unsigned character value.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains an unsigned character pointer.

Output

Name	Type	Description
uCharValuePtr	pointer to unsigned character	Unsigned character pointer `CA_VariantGetUCharPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantGetVariantPtr

```
HRESULT status = CA_VariantGetVariantPtr (VARIANT *variant,
                            VARIANT **variantPtr);
```

Purpose

Copies the value in a variant into a variant pointer variable.

`CA_VariantGetVariantPtr` returns an error if the variant does not contain a pointer to variant.

Parameters

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to a variant that contains a `VARIANT` pointer.

Output

Name	Type	Description
variantPtr	pointer to `VARIANT`	`VARIANT` pointer `CA_VariantGetVariantPtr` copies from the variant.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasArray

```
HRESULT status = CA_VariantHasArray (VARIANT *variant);
```

Purpose

Determines whether a variant contains an array. This is true if the type field of the variant contains the `VT_CAVT` modifier.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** type is an array. `0` if **variant** type is not an array. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasBool

```
HRESULT status = CA_VariantHasBool (VARIANT *variant);
```

Purpose

Determines whether a variant contains a VBOOL value.

Parameter

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	HRESULT	1 if the **variant** contains a VBOOL value. 0 if **variant** does not contain a VBOOL value. < 0 if **variant** is invalid. Refer to Table 11-18 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantHasBSTR

```
HRESULT status = CA_VariantHasBSTR (VARIANT *variant);
```

Purpose

Determines whether a variant contains a BSTR value.

Parameter

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	HRESULT	1 if the **variant** contains a BSTR value. 0 if **variant** does not contain a BSTR value. < 0 if **variant** is invalid. Refer to Table 11-18 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantHasCString

```
HRESULT status = CA_VariantHasCString (VARIANT *variant);
```

Purpose

Determines whether a variant contains a string. Variants do not contain C-style strings, but you can convert the strings they contain to C-style strings.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if the **variant** contains a string value. `0` if **variant** does not contain a string value. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasCurrency

```
HRESULT status = CA_VariantHasCurrency (VARIANT *variant);
```

Purpose

Determines whether a variant contains a CURRENCY value.

Parameter

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	HRESULT	1 if the **variant** contains a CURRENCY value. 0 if **variant** does not contain a CURRENCY value. < 0 if **variant** is invalid. Refer to Table 11-18 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantHasDate

```
HRESULT status = CA_VariantHasDate (VARIANT *variant);
```

Purpose

Determines whether a variant contains a DATE value.

Parameter

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	HRESULT	1 if **variant** contains a DATE value. 0 if **variant** does not contain a DATE value. < 0 if **variant** is invalid. Refer to Table 11-18 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantHasDispatch

```
HRESULT status = CA_VariantHasDispatch (VARIANT *variant);
```

Purpose

Determines whether a variant contains an IDispatch interface.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains an IDispatch interface. `0` if **variant** does not contain an IDispatch interface. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasDouble

```
HRESULT status = CA_VariantHasDouble (VARIANT *variant);
```

Purpose

Determines whether a variant contains a double-precision value.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains a double-precision value. `0` if **variant** does not contain a double-precision value. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasError

```
HRESULT status = CA_VariantHasError (VARIANT *variant);
```

Purpose

Determines whether a variant contains an SCODE value. SCODE is the data type for an error value.

Parameter

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	HRESULT	1 if **variant** contains an SCODE value. 0 if **variant** does not contain an SCODE value. < 0 if **variant** is invalid. Refer to Table 11-18 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantHasFloat

```
HRESULT status = CA_VariantHasFloat (VARIANT *variant);
```

Purpose

Determines whether a variant contains a single-precision, floating-point value.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains a single-precision value. `0` if **variant** does not contain a single-precision value. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasInt

```
HRESULT status = CA_VariantHasInt (VARIANT *variant);
```

Purpose

Determines whether a variant contains an integer value.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains an integer value. `0` if **variant** does not contain an integer value. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasIUnknown

```
HRESULT status = CA_VariantHasIUnknown (VARIANT *variant);
```

Purpose

Determines whether a variant contains an IUnknown interface.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains an IUnknown interface. `0` if **variant** does not contain an IUnknown interface. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasLong

```
HRESULT status = CA_VariantHasLong (VARIANT *variant);
```

Purpose

Determines whether a variant contains a long integer value.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains a long integer value. `0` if **variant** does not contain a long integer value. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasNull

```
HRESULT status = CA_VariantHasNull (VARIANT *variant);
```

Purpose

Determines whether the data type of a variant is VT_NULL.

Parameter

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	HRESULT	1 if **variant** has a type of VT_NULL. 0 if **variant** does not have a type of VT_NULL. < 0 if **variant** is invalid. Refer to Table 11-18 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantHasObjHandle

```
HRESULT status = CA_VariantHasObjHandle (VARIANT *variant);
```

Purpose

Determines whether a variant contains a value that you can convert to a `CAObjHandle` using `CA_VariantGetObjHandle`. This is true if the variant type is `CAVT_DISPATCH` and the IDispatch pointer the variant contains is not `NULL`.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains a non-null IDispatch interface. `0` if **variant** does not contain a non-null IDispatch interface. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasPtr

```
HRESULT status = CA_VariantHasPtr (VARIANT *variant);
```

Purpose

Determines whether a variant contains a pointer to a data value. This is true if the type field of the variant contains the `VT_BYREF` flag.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains a pointer to a value. `0` if **variant** does not contain a pointer to a value. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasShort

```
HRESULT status = CA_VariantHasShort (VARIANT *variant);
```

Purpose

Determines whether a variant contains a short integer value.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains a short integer value. `0` if **variant** does not contain a short integer value. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantHasUChar

```
HRESULT status = CA_VariantHasUChar (VARIANT *variant);
```

Purpose

Determines whether a variant contains an unsigned character value.

Parameter

Input

Name	Type	Description
variant	pointer to `VARIANT`	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	`HRESULT`	`1` if **variant** contains an unsigned character value. `0` if **variant** does not contain an unsigned character value. `< 0` if **variant** is invalid. Refer to Table 11-18 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantInt

```
VARIANT variant = CA_VariantInt (int intValue);
```

Purpose

Converts an integer value to a variant that contains the integer value. Use `CA_VariantInt` to pass an integer value as a variant parameter.

Parameter

Input

Name	Type	Description
intValue	integer	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **intValue**.

CA_VariantIsEmpty

```
HRESULT status = CA_VariantIsEmpty (VARIANT *variant);
```

Purpose

Determines whether the data type of a variant type is VT_EMPTY.

Parameter

Input

Name	Type	Description
variant	pointer to VARIANT	Pointer to the variant variable to inspect.

Return Value

Name	Type	Description
status	HRESULT	1 if **variant** has a type of VT_EMPTY. 0 if **variant** does not have a type of VT_EMPTY. < 0 if **variant** is invalid. Refer to Table 11-18 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantIUnknown

```
VARIANT variant = CA_VariantIUnknown (LPUNKNOWN IUnknownValue);
```

Purpose

Converts an `LPUNKNOWN` value to a variant that contains an `LPUNKNOWN` value. Use `CA_VariantIUnknown` to pass an `LPUNKNOWN` value as a variant parameter.

Parameter

Input

Name	Type	Description
IUnknownValue	`LPUNKNOWN`	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **IUnknownValue**.

CA_VariantLong

```
VARIANT variant = CA_VariantLong (long longValue);
```

Purpose

Converts a long integer value to a variant that contains the long integer value. Use `CA_VariantLong` to pass a long integer value as a variant parameter.

Parameter

Input

Name	Type	Description
longValue	long integer	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **longValue**.

CA_VariantNULL

```
VARIANT variant = CA_VariantNULL (void);
```

Purpose

Returns a variant in which the value of the type field is `VT_NULL`. Use `CA_VariantNULL` to pass a `NULL` variant as a parameter.

Parameters

None.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant in which LabWindows/CVI sets the type field to `VT_NULL`.

CA_VariantSet1DArray

```
HRESULT status = CA_VariantSet1DArray (VARIANT *variant,
                            unsigned int arrayType, unsigned int numElements,
                            void *array);
```

Purpose

Creates a safe array from a 1D array and stores the safe array in a variant.

Parameters

Input/Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSet1DArray` assigns the safe array.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of **array**.
numElements	unsigned integer	Number of elements in **array**.
array	depends on value of **arrayType**	1D array that `CA_VariantSet1DArray` converts to a safe array.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter can contain any of the fundamental types in Table 11-2 except `CAVT_EMPTY`, `CAVT_NULL`, and `CAVT_OBJHANDLE`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

Note `CA_VariantSet1DArray` ***does not make copies of*** `BSTR`, `VARIANT`, `LPUNKNOWN`, ***or*** `LPDISPATCH` ***elements. It simply copies the pointers into the created safe array. Therefore, when you call*** `CA_VariantClear`, ***which frees the safe array and all its contents, the*** `BSTR`, `VARIANT`, `LPUNKNOWN`, ***or*** `LPDISPATCH` ***elements of the input array parameter become invalid.***

CA_VariantSet2DArray

```
HRESULT status = CA_VariantSet2DArray (VARIANT *variant,
                          unsigned int arrayType,
                          unsigned int numElemsDim1,
                          unsigned int numElemsDim2, void *array);
```

Purpose

Creates a safe array from a 2D array and stores the safe array in a variant.

Parameters

Input/Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSet2DArray` assigns the safe array.

Input

Name	Type	Description
arrayType	unsigned integer	Data type of **array**.
numElemsDim1	unsigned integer	Number of elements in the first dimension of **array**.
numElemsDim2	unsigned integer	Number of elements in the second dimension of **array**.
array	depends on value of **arrayType**	2D array that `CA_VariantSet2DArray` converts to a safe array.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter can contain any of the fundamental types in Table 11-2 except `CAVT_EMPTY`, `CAVT_NULL`, and `CAVT_OBJHANDLE`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

Note `CA_VariantSet2DArray` ***does not make copies of*** `BSTR`, `VARIANT`, `LPUNKNOWN`, ***or*** `LPDISPATCH` ***elements. It simply copies the pointers into the created safe array. Therefore, when you call*** `CA_VariantClear`, ***which frees the safe array and all its contents, the*** `BSTR`, `VARIANT`, `LPUNKNOWN`, ***or*** `LPDISPATCH` ***elements of the input array parameter become invalid.***

CA_VariantSetBool

```
HRESULT status = CA_VariantSetBool (VARIANT *variant, VBOOL boolValue);
```

Purpose

Stores a `VBOOL` value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
boolValue	`VBOOL`	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetBool` assigns **boolValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetBoolPtr

```
HRESULT status = CA_VariantSetBoolPtr (VARIANT *variant,
                          VBOOL *boolValuePtr);
```

Purpose

Stores a pointer to a VBOOL value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
boolValuePtr	pointer to VBOOL	Pointer to store in **variant**.

Output

Name	Type	Description
variant	VARIANT	Variant to which CA_VariantSetBoolPtr assigns **boolValuePtr**.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantSetBSTR

```
HRESULT status = CA_VariantSetBSTR (VARIANT *variant, BSTR BSTRValue);
```

Purpose

Stores a BSTR in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
BSTRValue	BSTR	Value to store in **variant**.

Output

Name	Type	Description
variant	VARIANT	Variant to which CA_VariantSetBSTR assigns **BSTRValue**.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantSetBSTRPtr

```
HRESULT status = CA_VariantSetBSTRPtr (VARIANT *variant,
                                       BSTR *BSTRValuePtr);
```

Purpose

Stores a pointer to a `BSTR` in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
BSTRValuePtr	pointer to `BSTR`	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetBSTRPtr` assigns **BSTRValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetCString

```
HRESULT status = CA_VariantSetCString (VARIANT *variant, char *cString);
```

Purpose

Converts a C-style string to a BSTR, stores the BSTR in a variant, and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
cString	string	Value CA_VariantSetCString converts to a BSTR.

Output

Name	Type	Description
variant	VARIANT	Variant to which CA_VariantSetCString assigns the BSTR it creates.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantSetCurrency

```
HRESULT status = CA_VariantSetCurrency (VARIANT *variant,
                                        CURRENCY currencyValue);
```

Purpose

Stores a `CURRENCY` value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
currencyValue	`CURRENCY`	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetCurrency` assigns **currencyValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetCurrencyPtr

```
HRESULT status = CA_VariantSetCurrencyPtr (VARIANT *variant,
                            CURRENCY *currencyValuePtr);
```

Purpose

Stores a pointer to a `CURRENCY` value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
currencyValuePtr	pointer to `CURRENCY`	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetCurrencyPtr` assigns **currencyValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetDate

```
HRESULT status = CA_VariantSetDate (VARIANT *variant, DATE dateValue);
```

Purpose

Stores a DATE value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
dateValue	DATE	Value to store in **variant**.

Output

Name	Type	Description
variant	VARIANT	Variant to which CA_VariantSetDate assigns **dateValue**.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantSetDatePtr

```
HRESULT status = CA_VariantSetDatePtr (VARIANT *variant,
                                       DATE *dateValuePtr);
```

Purpose

Stores a pointer to a DATE value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
dateValuePtr	pointer to DATE	Pointer to store in **variant**.

Output

Name	Type	Description
variant	VARIANT	Variant to which CA_VariantSetDatePtr assigns **dateValuePtr**.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantSetDispatch

```
HRESULT status = CA_VariantSetDispatch (VARIANT *variant,
                          LPDISPATCH dispatchValue);
```

Purpose

Stores an `LPDISPATCH` value in a variant and sets the type field of the variant accordingly. An `LPDISPATCH` value is a dispatch pointer for an ActiveX Automation object interface.

Parameters

Input

Name	Type	Description
dispatchValue	`LPDISPATCH`	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetDispatch` assigns **dispatchValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetDispatchPtr

```
HRESULT status = CA_VariantSetDispatchPtr (VARIANT *variant,
                           LPDISPATCH *dispatchValuePtr);
```

Purpose

Stores a pointer to an `LPDISPATCH` value in a variant and sets the type field of the variant accordingly. An `LPDISPATCH` value is a dispatch pointer for an ActiveX Automation object interface.

Parameters

Input

Name	Type	Description
dispatchValuePtr	pointer to `LPDISPATCH`	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetDispatchPtr` assigns **dispatchValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetDouble

```
HRESULT status = CA_VariantSetDouble (VARIANT *variant,
                                      double doubleValue);
```

Purpose

Stores a double-precision value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
doubleValue	double-precision	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetDouble` assigns **doubleValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetDoublePtr

```
HRESULT status = CA_VariantSetDoublePtr (VARIANT *variant,
                                         double *doubleValuePtr);
```

Purpose

Stores a pointer to a double-precision value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
doubleValuePtr	pointer to double-precision	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetDoublePtr` assigns **doubleValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetEmpty

```
HRESULT status = CA_VariantSetEmpty (VARIANT *variant);
```

Purpose

Marks a variant as empty by setting its type field to `VT_EMPTY`.

Parameter

Output

Name	Type	Description
variant	`VARIANT`	Variant to mark as empty.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetError

```
HRESULT status = CA_VariantSetError (VARIANT *variant, SCODE errorValue);
```

Purpose

Stores an `SCODE` value in a variant and sets the type field of the variant accordingly. `SCODE` is the data type for an error value.

Parameters

Input

Name	Type	Description
errorValue	`SCODE`	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetError` assigns **errorValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetErrorPtr

```
HRESULT status = CA_VariantSetErrorPtr (VARIANT *variant,
                                        SCODE *errorValuePtr);
```

Purpose

Stores a pointer to an `SCODE` value in a variant and sets the type field of the variant accordingly. `SCODE` is the data type for an error value.

Parameters

Input

Name	Type	Description
errorValuePtr	pointer to `SCODE`	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetErrorPtr` assigns **errorValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetFloat

```
HRESULT status = CA_VariantSetFloat (VARIANT *variant, float floatValue);
```

Purpose

Stores a single-precision, floating-point value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
floatValue	single-precision	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetFloat` assigns **floatValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetFloatPtr

```
HRESULT status = CA_VariantSetFloatPtr (VARIANT *variant,
                                        float *floatValuePtr);
```

Purpose

Stores a pointer to a single-precision, floating-point value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
floatValuePtr	pointer to single-precision	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetFloatPtr` assigns **floatValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetInt

```
HRESULT status = CA_VariantSetInt (VARIANT *variant, int intValue);
```

Purpose

Stores an integer value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
intValue	integer	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetInt` assigns **intValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetIntPtr

```
HRESULT status = CA_VariantSetIntPtr (VARIANT *variant, int *intValuePtr);
```

Purpose

Stores a pointer to an integer value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
intValuePtr	pointer to integer	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetIntPtr` assigns **intValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetIUnknown

```
HRESULT status = CA_VariantSetIUnknown (VARIANT *variant,
                            LPUNKNOWN IUnknownValue);
```

Purpose

Stores an `LPUNKNOWN` value in a variant and sets the type field of the variant accordingly. An `LPUNKNOWN` value is a pointer to an unknown interface.

Parameters

Input

Name	Type	Description
IUnknownValue	`LPDISPATCH`	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetIUnknown` assigns **IUnknownValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetIUnknownPtr

```
HRESULT status = CA_VariantSetIUnknownPtr (VARIANT *variant,
                          LPUNKNOWN *IUnknownValuePtr);
```

Purpose

Stores a pointer to an `LPUNKNOWN` value in a variant and sets the type field of the variant accordingly. An `LPUNKNOWN` value is a pointer to an unknown interface.

Parameters

Input

Name	Type	Description
IUnknownValuePtr	pointer to `LPUNKNOWN`	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetIUnknownPtr` assigns **IUnknownValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetLong

```
HRESULT status = CA_VariantSetLong (VARIANT *variant, long longValue);
```

Purpose

Stores a long integer value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
longValue	long integer	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetLong` assigns **longValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetLongPtr

```
HRESULT status = CA_VariantSetLongPtr (VARIANT *variant,
                                       long *longValuePtr);
```

Purpose

Stores a pointer to a long integer value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
longValuePtr	pointer to long integer	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetLongPtr` assigns **longValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetNULL

```
HRESULT status = CA_VariantSetNULL (VARIANT *variant);
```

Purpose

Marks a variant as NULL by setting its type field to VT_NULL.

Parameter

Output

Name	Type	Description
variant	VARIANT	Variant to mark as NULL.

Return Value

Name	Type	Description
status	HRESULT	Refer to Table 11-19 for error codes or to cvi\sdk\winerror.h for Windows SDK error codes.

CA_VariantSetSafeArray

```
HRESULT status = CA_VariantSetSafeArray (VARIANT *variant,
                          unsigned int arrayType, LPSAFEARRAY safeArray);
```

Purpose

Stores a safe array in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
arrayType	unsigned integer	Type of the safe array.
safeArray	`LPSAFEARRAY`	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetSafeArray` assigns **safeArray**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter can be any of the fundamental types in Table 11-2 except `CAVT_EMPTY`, `CAVT_NULL`, or `CAVT_OBJHANDLE`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

CA_VariantSetSafeArrayPtr

```
HRESULT status = CA_VariantSetSafeArrayPtr (VARIANT *variant,
                            unsigned int arrayType,
                            LPSAFEARRAY *safeArrayPtr);
```

Purpose

Stores a pointer to a safe array in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
arrayType	unsigned integer	Type of the safe array.
safeArrayPtr	pointer to `LPSAFEARRAY`	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetSafeArrayPtr` assigns **safeArrayPtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

Parameter Discussion

The **arrayType** parameter can be any of the fundamental types in Table 11-2 except `CAVT_EMPTY`, `CAVT_NULL`, or `CAVT_OBJHANDLE`. Table 11-2 is in the *Data Types for Variants, Safe Arrays, and Properties* section of the *ActiveX Automation Library Function Overview* section of this chapter.

CA_VariantSetShort

```
HRESULT status = CA_VariantSetShort (VARIANT *variant, short shortValue);
```

Purpose

Stores a short integer value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
shortValue	short integer	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetShort` assigns **shortValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetShortPtr

```
HRESULT status = CA_VariantSetShortPtr (VARIANT *variant,
                                        short *shortValuePtr);
```

Purpose

Stores a pointer to a short integer value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
shortValuePtr	pointer to short integer	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetShortPtr` assigns **shortValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetUChar

```
HRESULT status = CA_VariantSetUChar (VARIANT *variant,
                                     unsigned char uCharValue);
```

Purpose

Stores an unsigned character value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
uCharValue	unsigned character	Value to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetUChar` assigns **uCharValue**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetUCharPtr

```
HRESULT status = CA_VariantSetUCharPtr (VARIANT *variant,
                            unsigned char *uCharValuePtr);
```

Purpose

Stores a pointer to an unsigned character value in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
uCharValuePtr	pointer to unsigned character	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetUCharPtr` assigns **uCharValuePtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantSetVariantPtr

```
HRESULT status = CA_VariantSetVariantPtr (VARIANT *variant,
                          VARIANT *variantPtr);
```

Purpose

Stores a pointer to a variant in a variant and sets the type field of the variant accordingly.

Parameters

Input

Name	Type	Description
variantPtr	pointer to `VARIANT`	Pointer to store in **variant**.

Output

Name	Type	Description
variant	`VARIANT`	Variant to which `CA_VariantSetVariantPtr` assigns **variantPtr**.

Return Value

Name	Type	Description
status	`HRESULT`	Refer to Table 11-19 for error codes or to `cvi\sdk\winerror.h` for Windows SDK error codes.

CA_VariantShort

```
VARIANT variant = CA_VariantShort (short shortValue);
```

Purpose

Converts a short integer value to a variant that contains the short integer value. Use `CA_VariantShort` to pass a short integer value as a `VARIANT` parameter.

Parameter

Input

Name	Type	Description
shortValue	short integer	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **shortValue**.

CA_VariantUChar

```
VARIANT variant = CA_VariantUChar (unsigned char uCharValue);
```

Purpose

Converts an unsigned character value to a variant that contains the unsigned character value. Use `CA_VariantUChar` to pass an unsigned character value as a `VARIANT` parameter.

Parameter

Input

Name	Type	Description
uCharValue	unsigned character	Value to store in the variant.

Return Value

Name	Type	Description
variant	`VARIANT`	Variant that contains the value in **uCharValue**.

Error Conditions

Most of the functions in the ActiveX Automation Library return error codes, which are defined in `cvi\include\cviauto.h` or in `cvi\sdk\include\winerror.h`. Table 11-19 lists error codes defined in `cviauto.h` or `winerror.h` that the ActiveX Automation Library returns explicitly. Occasionally, an ActiveX Automation Library function returns an error code because an internal call to a Windows function returns it. Table 11-19 does not list such error codes, but `winerror.h` does.

You can use `CA_GetAutomationErrorString` to get the description of an error code. You can use `CA_DisplayErrorInfo` to display the description of an error code.

Table 11-19. ActiveX Automation Library Error Codes

Defined Constant	Value	Description
`E_CVIAUTO_NO_ERROR`	`0`	No error.
`E_CVIAUTO_INVALID_TYPE_DESC`	`0X80040201`	Type you passed is an invalid Automation data type.
`E_CVIAUTO_INVALID_RETURN_TYPE`	`0X80040202`	Type you passed is an invalid return type.
`E_CVIAUTO_USE_CAVT_TYPE_DESC`	`0X80040203`	Use `CAVT_` constants for data types instead of `VT_` constants.
`E_CVIAUTO_INVALID_NUM_DIM`	`0X80040204`	Number of dimensions in the safe array does not match the number this function requires.
`E_CVIAUTO_DIFF_SAFEARRAY_TYPE`	`0X80040205`	Safe array type does not match the type you request.
`E_CVIAUTO_VARIANT_NOT_SAFEARRAY`	`0X80040206`	Variant does not contain a safe array.
`E_CVIAUTO_NULL_RET_VAL_PARAM`	`0X80040207`	Return value parameter must not be `NULL`.
`E_CVIAUTO_DLL_LOAD_FAILED`	`0X80040208`	Could not load the Automation Support DLL.

Table 11-19. ActiveX Automation Library Error Codes (Continued)

Defined Constant	Value	Description
E_CVIAUTO_BAD_DLL_VERSION	0X80040209	Automation Support DLL version does not match the import library.
E_CVIAUTO_COULD_NOT_CREATE_MUTEX	0X8004020A	Unable to create a required mutex.
DISP_E_TYPEMISMATCH	0X80020005	Variant type does not match the type this function requires.
DISP_E_BADVARTYPE	0X80020008	Variant type is invalid for the operation you request.
DISP_E_EXCEPTION	0X80020009	Exception occurred in server; refer to ERRORINFO structure for details or call CA_DisplayErrorInfo.
E_HANDLE	0X80070006	Invalid object handle.
E_OUTOFMEMORY	0X8007000E	Out of memory.
E_INVALIDARG	0X80070057	Invalid argument, such as NULL pointer.
ERROR_INSUFFICIENT_BUFFER	0X8007007A	Array buffer is not large enough to receive the contents of the safe array.

Note ***Refer to*** `cvi\sdk\winerror.h` ***for more Windows SDK error codes.***

A

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently 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: _____

_____

_____

_____

_____

LabWindows/CVI 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

Hardware revision _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

National Instruments software _____

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: *LabWindows/CVI Standard Libraries Reference Manual*

Edition Date: February 1998

Part Number: 320682D-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

Glossary

Prefix	Meanings	Value
n-	nano-	10^{-9}
µ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}

Numbers/Symbols

1D — One-dimensional.

2D — Two-dimensional.

A

A/D — Analog-to-digital.

AI — Analog input.

analog trigger — A trigger that occurs at a user-selected point on an incoming analog signal. You can set triggering to occur at a specific level on an increasing or a decreasing signal, that is, a positive or negative slope. You can implement analog triggering in software or in hardware. When you implement it in software, all data is collected, transferred into system memory, and analyzed for the trigger condition. When you implement it in hardware, no data is transferred to system memory until the trigger condition has occurred.

ANSI — American National Standards Institute.

AO — Analog output.

asynchronous — (1) Hardware—A property of an event that occurs at an arbitrary time, without synchronization to a reference clock.

(2) Software—A property of a function that begins an operation and returns before the completion or termination of the operation.

automatic serial — A feature in which the GPIB polling driver automatically executes serial polls whenever a device asserts the SRQ line.

B

B — Bytes.

C

C locale — The minimal environment for compiling a C program.

counter/timer — A circuit that counts external pulses or clock pulses (timing).

coupling — The manner in which a signal is connected from one location to another.

D

Data acquisition — (1) Collecting and measuring electrical signals from sensors, transducers, and test probes or fixtures and inputting them to a computer for processing.

(2) Collecting and measuring the same kinds of electrical signals with A/D and/or DIO boards plugged into a PC and possibly generating control signals with D/A and/or DIO boards in the same PC.

device — Refers to a DAQ device inside your computer or attached directly to your computer through a parallel port. Plug-in boards, PCMCIA cards, and devices such as the DAQPad-1200, which connects to your computer parallel port, are all examples of DAQ devices. SCXI modules are distinct from devices with the exception of the SCXI-1200, which is a hybrid.

differential input — An analog input that consists of two terminals, both of which are isolated from computer ground, whose difference is measured.

digital port — Refer to port.

DIO — Digital I/O.

F

FIFO — A first-in first-out memory buffer; the first data stored is the first data sent to the acceptor.

format string — A mini-program that instructs the formatting and scanning functions how to transform the input arguments to the output arguments. For conciseness, format strings are constructed using single-character codes.

G

gender — Refers to cable connector types. A male connector is one with protruding pins, like a lamp plug. A female connector has holes, like an outlet.

gender changer — A small device you can attach to serial cable connectors or PC sockets, among others, to convert a female connector into a male or a male connector into a female.

GPIB — General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standards 488.1-1987 and 488.2-1992.

group — A collection of digital ports, combined to form a larger entity for digital input and/or output.

H

handshaking — Prevents overflow of the input queue that occurs when the receiver is unable to empty its input queue as quickly as the sender is able to fill it. The RS-232 Library has two types of handshaking: software handshaking and hardware handshaking. You should enable one or the other if you want to ensure that your application program synchronizes its data transfers with other serial devices that perform handshaking.

Hz — Hertz.

I

ID	Identification.
IEEE	Institute of Electrical and Electronics Engineers.
Instrument Library	A LabWindows/CVI library that contains instrument drivers.
interrupt	A computer signal that indicates the CPU should suspend its current task to service a designated activity.
I/O	Input/output.

K

KB	Kilobytes of memory.

M

MB	Megabytes of memory.
MIO	Multifunction I/O.
ms	Milliseconds.

N

NI-488 functions	National Instruments functions you use to communicate with GPIB devices built according to the ANSI/IEEE Standards 488.1-1987 and 488.2-1992.
NI-488.2 routines	National Instruments routines you use to communicate with GPIB devices built according to the ANSI/IEEE Standard 488.2-1992.

P

port	A digital port that consists of four or eight lines of digital input and/or output.

R

resolution — The smallest signal increment a measurement system can detect. Resolution can be expressed in bits, in proportions, or in percent of full scale. For example, a system has 12-bit resolution, one part in 4,096 resolution, and 0.0244 percent of full scale.

S

SCXI — Signal Conditioning eXtensions for Instrumentation; the National Instruments product line for conditioning low-level signals within an external chassis near sensors so only high-level signals are sent to DAQ boards in the noisy PC environment.

software trigger — A programmed event that triggers an event such as data acquisition.

standard libraries — The LabWindows/CVI Analysis, ANSI C, DDE, Formatting and I/O, GPIB and GPIB-488.2, RS-232, TCP, and Utility libraries.

synchronous — (1) Hardware—Property of an event that is synchronized to a reference clock.

(2) Software—Property of a function that begins an operation and returns only when the operation is complete.

T

TC — Terminal count.

TTL — Transistor-Transistor Logic.

X

Xmodem functions — Allow you to transfer files that use a data transfer protocol. The protocol uses a generally accepted technique to perform serial file transfers with error-checking. Files are sent in packets that contain data from the files plus error-checking and synchronization information.

Index

Numbers and Special Characters

1D array functions. *See* one-dimensional array operation functions.
1D complex operation functions. *See* one-dimensional complex operation functions.
2D array functions. *See* two-dimensional array operation functions.
* (asterisks) in format specifiers
    formatting functions, 2-61
    scanning functions, 2-74

A

Abs1D function, 3-5
accessing physical memory. *See* physical memory access functions.
accessing window properties. *See* window properties, accessing.
ActiveX Automation Library
    data type modifiers for variants, safe arrays, and properties (table), 11-10
    data types for variants, safe arrays, and properties (table), 11-9 to 11-10
    error conditions, 11-202 to 11-203
    events not supported, 11-2
    function panels
        classes, 11-7 to 11-8
        function tree, 11-3 to 11-7
    function reference
        CA_Array1DToSafeArray, 11-12
        CA_Array2DToSafeArray, 11-13 to 11-14
        CA_BSTRGetCString, 11-15
        CA_BSTRGetCStringBuf, 11-16
        CA_BSTRGetCStringLen, 11-17
        CA_CreateObjectByClassId, 11-18 to 11-19
        CA_CreateObjectByProgId, 11-20 to 11-21
        CA_CreateObjHandleFromIDispatch, 11-22
        CA_CStringToBSTR, 11-23
        CA_DefaultValueVariant, 11-8, 11-24
        CA_DiscardObjHandle, 11-25
        CA_DisplayErrorInfo, 11-26
        CA_FreeMemory, 11-27
        CA_FreeUnusedServers, 11-28
        CA_GetActiveObjectByClassId, 11-29 to 11-30
        CA_GetActiveObjectByProgId, 11-31 to 11-32
        CA_GetAutomationErrorString, 11-33
        CA_GetDispatchFromObjHandle, 11-34
        CA_GetLocale, 11-35
        CA_InvokeHelper, 11-36 to 11-40
        CA_InvokeHeplerV, 11-41
        CA_LoadObjectFromFile, 11-42 to 11-43
        CA_LoadObjectFromFileByClassId, 11-44 to 11-45
        CA_LoadObjectFromFileByProgId, 11-46 to 11-47
        CA_MethodInvoke, 11-48 to 11-49
        CA_MethodInvokeV, 11-50
        CA_PropertyGet, 11-51 to 11-52
        CA_PropertySet, 11-53 to 11-54
        CA_PropertySetByRef, 11-55 to 11-56
        CA_PropertySetByRefV, 11-57
        CA_PropertySetV, 11-58
        CA_SafeArrayDestroy, 11-59
        CA_SafeArrayGet1DSize, 11-60
        CA_SafeArrayGet2DSize, 11-61
        CA_SafeArrayGetNumDims, 11-62

CA_SafeArrayTo1DArray, 11-63 to 11-65
CA_SafeArrayTo1DArrayBuf, 11-66 to 11-68
CA_SafeArrayTo2DArray, 11-69 to 11-71
CA_SafeArrayTo2DArrayBuf, 11-72 to 11-74
CA_SetLocale, 11-75 to 11-76
CA_VariantBool, 11-77
CA_VariantBSTR, 11-78
CA_VariantClear, 11-8, 11-9, 11-79
CA_VariantConvertToType, 11-9, 11-80 to 11-81
CA_VariantCopy, 11-82
CA_VariantCurrency, 11-83
CA_VariantDate, 11-84
CA_VariantDispatch, 11-85
CA_VariantDouble, 11-86
CA_VariantEmpty, 11-87
CA_VariantError, 11-88
CA_VariantFloat, 11-89
CA_VariantGet1DArray, 11-90 to 11-92
CA_VariantGet1DArrayBuf, 11-93 to 11-95
CA_VariantGet1DArraySize, 11-96
CA_VariantGet2DArray, 11-97 to 11-99
CA_VariantGet2DArrayBuf, 11-100 to 11-102
CA_VariantGet2DArraySize, 11-103
CA_VariantGetArrayNumDims, 11-104
CA_VariantGetBool, 11-105
CA_VariantGetBoolPtr, 11-106
CA_VariantGetBSTR, 11-107
CA_VariantGetBSTRPtr, 11-108
CA_VariantGetCString, 11-109
CA_VariantGetCStringBuf, 11-110
CA_VariantGetCStringLen, 11-111
CA_VariantGetCurrency, 11-112
CA_VariantGetCurrencyPtr, 11-113
CA_VariantGetDate, 11-114
CA_VariantGetDatePtr, 11-115
CA_VariantGetDispatch, 11-116
CA_VariantGetDispatchPtr, 11-117
CA_VariantGetDouble, 11-118
CA_VariantGetDoublePtr, 11-119
CA_VariantGetError, 11-120
CA_VariantGetErrorPtr, 11-121
CA_VariantGetFloat, 11-122
CA_VariantGetFloatPtr, 11-123
CA_VariantGetInt, 11-124
CA_VariantGetIntPtr, 11-125
CA_VariantGetIUnknown, 11-126
CA_VariantGetIUnknownPtr, 11-127
CA_VariantGetLong, 11-9, 11-128
CA_VariantGetLongPtr, 11-129
CA_VariantGetObjHandle, 11-130
CA_VariantGetSafeArray, 11-131 to 11-132
CA_VariantGetSafeArrayPtr, 11-133 to 11-134
CA_VariantGetShort, 11-135
CA_VariantGetShortPtr, 11-136
CA_VariantGetType, 11-8, 11-137
CA_VariantGetUChar, 11-138
CA_VariantGetUChartPtr, 11-139
CA_VariantGetVariantPtr, 11-140
CA_VariantHasArray, 11-141
CA_VariantHasBool, 11-142
CA_VariantHasBSTR, 11-143
CA_VariantHasCString, 11-144
CA_VariantHasCurrency, 11-145
CA_VariantHasDate, 11-146
CA_VariantHasDispatch, 11-147
CA_VariantHasDouble, 11-148
CA_VariantHasError, 11-149
CA_VariantHasFloat, 11-150
CA_VariantHasInt, 11-151

CA_VariantHasIUnknown, 11-152
CA_VariantHasLong, 11-9, 11-153
CA_VariantHasNull, 11-154
CA_VariantHasObjectHandle, 11-155
CA_VariantHasPtr, 11-156
CA_VariantHasShort, 11-9, 11-157
CA_VariantHasUChar, 11-158
CA_VariantInt, 11-159
CA_VariantIsEmpty, 11-160
CA_VariantIUnknown, 11-161
CA_VariantLong, 11-162
CA_VariantNULL, 11-163
CA_VariantSet1DArray, 11-164 to 11-165
CA_VariantSet2DArray, 11-166 to 11-167
CA_VariantSetBool, 11-168
CA_VariantSetBoolPtr, 11-169
CA_VariantSetBSTR, 11-170
CA_VariantSetBSTRPtr, 11-171
CA_VariantSetCString, 11-172
CA_VariantSetCurrency, 11-173
CA_VariantSetCurrencyPtr, 11-174
CA_VariantSetDate, 11-175
CA_VariantSetDatePtr, 11-176
CA_VariantSetDispatch, 11-177
CA_VariantSetDispatchPtr, 11-178
CA_VariantSetDouble, 11-179
CA_VariantSetDoublePtr, 11-180
CA_VariantSetEmpty, 11-181
CA_VariantSetError, 11-182
CA_VariantSetErrorPtr, 11-183
CA_VariantSetFloat, 11-184
CA_VariantSetFloatPtr, 11-185
CA_VariantSetInt, 11-186
CA_VariantSetIntPtr, 11-187
CA_VariantSetIUnknown, 11-188
CA_VariantSetIUnknownPtr, 11-189
CA_VariantSetLong, 11-190
CA_VariantSetLongPtr, 11-191
CA_VariantSetNULL, 11-192
CA_VariantSetSafeArray, 11-193
CA_VariantSetSafeArrayPtr, 11-194
CA_VariantSetShort, 11-195
CA_VariantSetShortPtr, 11-196
CA_VariantSetUChar, 11-197
CA_VariantSetUCharPtr, 11-198
CA_VariantSetVariantPtr, 11-199
CA_VariantShort, 11-200
CA_VariantUChar, 11-201
handling dynamic memory variants hold, 11-11
input variant parameters, 11-8
output variant parameters, 11-8 to 11-9
overview, 11-1 to 11-2
variants and safe arrays, 11-2
variants marked as empty by retrieval functions, 11-9
Add1D function, 3-6
Add2D function, 3-7
AdviseDDEDataReady function, 6-7 to 6-9
AIAcquireTriggeredWaveforms function, 10-8 to 10-13
AIAcquireWaveforms function, 10-14 to 10-16
AICheckAcquisition function, 10-17
AIClearAcquisition function, 10-18
AIReadAcquisition function, 10-19 to 10-20
AISampleChannel function, 10-21 to 10-22
AISampleChannels function, 10-23 to 10-24
AIStartAcquisition function, 10-25 to 10-26
analog input functions, Easy I/O for DAQ Library
AIAcquireTriggeredWaveforms, 10-8 to 10-13
AIAcquireWaveforms, 10-14 to 10-16
AISampleChannel, 10-21 to 10-22
AISampleChannels, 10-23 to 10-24
channel string, 10-4 to 10-5

analog output functions, Easy I/O for DAQ Library
AOClearWaveforms, 10-27
AOGenerateWaveforms, 10-28 to 10-29
AOUpdateChannel, 10-30
AOUpdateChannels, 10-31
channel string, 10-6 to 10-7
Analysis Library functions
error conditions, 3-51
function panels
classes and subclasses, 3-3
function tree (table), 3-1 to 3-2
hints for using, 3-4
function reference
Abs1D, 3-5
Add1D, 3-6
Add2D, 3-7
Clear1D, 3-8
Copy1D, 3-9
CxAdd, 3-10
CxAdd1D, 3-11
CxDiv, 3-12
CxDiv1D, 3-13
CxLinEv1D, 3-14 to 3-15
CxMul, 3-16
CxMul1D, 3-17
CxRecip, 3-18
CxSub, 3-19
CxSub1D, 3-20
Determinant, 3-21
Div1D, 3-22
Div2D, 3-23
DotProduct, 3-24
GetAnalysisErrorString, 3-25
Histogram, 3-26 to 3-27
InvMatrix, 3-28
LinEv1D, 3-29
LinEv2D, 3-30
MatrixMul, 3-31 to 3-32
MaxMin1D, 3-33
MaxMin2D, 3-34 to 3-35
Mean, 3-36
Mul1D, 3-37
Mul2D, 3-38
Neg1D, 3-39
Set1D, 3-40
Sort, 3-41
StdDev, 3-42
Sub1D, 3-43
Sub2D, 3-44
Subset1D, 3-45
ToPolar, 3-46
ToPolar1D, 3-47
ToRect, 3-48
ToRect1D, 3-49
Transpose, 3-50
overview, 3-1
reporting analysis errors, 3-4
ANSI C Library
C locale, 1-3 to 1-6
information values (table), 1-3 to 1-4
LC_COLLATE, 1-6
LC_CTYPE, 1-5 to 1-6
LC_MONETARY, 1-5
LC_NUMERIC, 1-5
LC_TIME, 1-6
character processing, 1-6
classes (table), 1-1 to 1-2
control functions, 1-9 to 1-11
errno set by file I/O functions, 1-7
fdopen function, 1-12
input/output facilities, 1-6
low-level I/O functions, 1-2
mathematical functions, 1-7
standard language additions, 1-3 to 1-6
string processing, 1-6
time and date functions, 1-7 to 1-9
ANSI C macros, 1-3
AOClearWaveforms function, 10-27
AOGenerateWaveforms function, 10-28 to 10-29
AOUpdateChannel function, 10-30

AOUpdateChannels function, 10-31
array functions
ActiveX Automation Library
CA_Array1DToSafeArray, 11-12
CA_Array2DToSafeArray, 11-13 to 11-14
CA_SafeArrayGet1DSize, 11-60
CA_SafeArrayGet2DSize, 11-61
CA_SafeArrayGetNumDims, 11-62
CA_SafeArrayTo1DArray, 11-63 to 11-65
CA_SafeArrayTo1DArrayBuf, 11-66 to 11-68
CA_SafeArrayTo2DArray, 11-69 to 11-71
CA_SafeArrayTo2DArrayBuf, 11-72 to 11-74
Analysis Library
Abs1D, 3-5
Add1D, 3-6
Add2D, 3-7
Clear1D, 3-8
Copy1D, 3-9
Div1D, 3-22
Div2D, 3-23
LinEv1D, 3-29
LinEv2D, 3-30
MaxMin1D, 3-33
MaxMin2D, 3-34 to 3-35
Mul1D, 3-37
Mul2D, 3-38
Neg1D, 3-39
Set1D, 3-40
Sub1D, 3-43
Sub2D, 3-44
Subset1D, 3-45
ArrayToFile function, 2-5 to 2-7
asterisks (*) in format specifiers
formatting functions, 2-61
scanning functions, 2-74
asynchronous acquisition functions, Easy I/O for DAQ Library
AICheckAcquisition, 10-17
AIClearAcquisition, 10-18
AIReadAcquisition, 10-19 to 10-20
AIStartAcquisition, 10-25 to 10-26
PlotLastAIWaveformsPopup, 10-63
asynchronous callbacks
notification of SRQ and other GPIB events, 4-10
restrictions with ibNotify function, 4-21
automatic serial polling
compatibility, 4-8
hardware interrupts, 4-8
purpose and use, 4-7
RQS events
ibInstallCallback function, 4-16
ibNotify function, 4-20
SRQI events
ibInstallCallback function, 4-16
ibNotify function, 4-20

B

Beep function, 8-6
board control functions, GPIB, 4-2, 4-6 to 4-7
break on library error functions
DisableBreakOnLibraryErrors, 8-22
EnableBreakOnLibraryErrors, 8-27
GetBreakOnLibraryErrors, 8-31
GetBreakOnProtectionErrors, 8-32
SetBreakOnLibraryErrors, 8-109 to 8-110
SetBreakOnProtectionErrors, 8-111 to 8-112
Breakpoint function, 8-7
BroadcastDDEDataReady function, 6-10 to 6-11
BSTR functions
CA_BSTRGetCString, 11-15
CA_BSTRGetCStringBuf, 11-16

CA_BSTRGetCStringLen, 11-17
CA_CStringToBSTR, 11-23
bus control functions, GPIB Library, 4-2
byte count variable (ibcntl), 4-6

C

C locale, 1-3 to 1-6
information values (table), 1-3 to 1-4
LC_COLLATE, 1-6
LC_CTYPE, 1-5 to 1-6
LC_MONETARY, 1-5
LC_NUMERIC, 1-5
LC_TIME, 1-6
CA_Array1DToSafeArray function, 11-12
CA_Array2DToSafeArray function, 11-13 to 11-14
CA_BSTRGetCString function, 11-15
CA_BSTRGetCStringBuf function, 11-16
CA_BSTRGetCStringLen function, 11-17
CA_CreateObjectByClassId function, 11-18 to 11-19
CA_CreateObjectByProgId function, 11-20 to 11-21
CA_CreateObjHandleFromIDispatch function, 11-22
CA_CStringToBSTR function, 11-23
CA_DEFAULT_VAL macro, 11-8
CA_DefaultValueVariant function, 11-8, 11-24
CA_DiscardObjHandle function, 11-25
CA_DisplayErrorInfo function, 11-26
CA_FreeMemory function, 11-27
CA_FreeUnusedServers function, 11-28
CA_GetActiveObjectByClassId function, 11-29 to 11-30
CA_GetActiveObjectByProgId function, 11-31 to 11-32
CA_GetAutomationErrorString function, 11-33
CA_GetDispatchFromObjHandle function, 11-34
CA_GetLocale function, 11-35
CA_InvokeHelper function, 11-36 to 11-40
CA_InvokeHeplerV function, 11-41
CA_LoadObjectFromFile function, 11-42 to 11-43
CA_LoadObjectFromFileByClassId function, 11-44 to 11-45
CA_LoadObjectFromFileByProgId function, 11-46 to 11-47
CA_MethodInvoke function, 11-48 to 11-49
CA_MethodInvokeV function, 11-50
CA_PropertyGet function, 11-51 to 11-52
CA_PropertySet function, 11-53 to 11-54
CA_PropertySetByRef function, 11-55 to 11-56
CA_PropertySetByRefV function, 11-57
CA_PropertySetV function, 11-58
CA_SafeArrayDestroy function, 11-59
CA_SafeArrayGet1DSize function, 11-60
CA_SafeArrayGet2DSize function, 11-61
CA_SafeArrayGetNumDims function, 11-62
CA_SafeArrayTo1DArray function, 11-63 to 11-65
CA_SafeArrayTo1DArrayBuf function, 11-66 to 11-68
CA_SafeArrayTo2DArray function, 11-69 to 11-71
CA_SafeArrayTo2DArrayBuf function, 11-72 to 11-74
CA_SetLocale function, 11-75 to 11-76
CA_VariantBool function, 11-77
CA_VariantBSTR function, 11-78
CA_VariantClear function, 11-8, 11-9, 11-79
CA_VariantConvertToType function, 11-9, 11-80 to 11-81
CA_VariantCopy function, 11-82
CA_VariantCurrency function, 11-83
CA_VariantDate function, 11-84
CA_VariantDispatch function, 11-85

CA_VariantDouble function, 11-86
CA_VariantEmpty function, 11-87
CA_VariantError function, 11-88
CA_VariantFloat function, 11-89
CA_VariantGet1DArray function, 11-90 to 11-92
CA_VariantGet1DArrayBuf function, 11-93 to 11-95
CA_VariantGet1DArraySize function, 11-96
CA_VariantGet2DArray function, 11-97 to 11-99
CA_VariantGet2DArrayBuf function, 11-100 to 11-102
CA_VariantGet2DArraySize function, 11-103
CA_VariantGetArrayNumDims function, 11-104
CA_VariantGetBool function, 11-105
CA_VariantGetBoolPtr function, 11-106
CA_VariantGetBSTR function, 11-107
CA_VariantGetBSTRPtr function, 11-108
CA_VariantGetCString function, 11-109
CA_VariantGetCStringBuf function, 11-110
CA_VariantGetCStringLen function, 11-111
CA_VariantGetCurrency function, 11-112
CA_VariantGetCurrencyPtr function, 11-113
CA_VariantGetDate function, 11-114
CA_VariantGetDatePtr function, 11-115
CA_VariantGetDispatch function, 11-116
CA_VariantGetDispatchPtr function, 11-117
CA_VariantGetDouble function, 11-118
CA_VariantGetDoublePtr function, 11-119
CA_VariantGetError function, 11-120
CA_VariantGetErrorPtr function, 11-121
CA_VariantGetFloat function, 11-122
CA_VariantGetFloatPtr function, 11-123
CA_VariantGetInt function, 11-124
CA_VariantGetIntPtr function, 11-125
CA_VariantGetIUnknown function, 11-126
CA_VariantGetIUnknownPtr function, 11-127
CA_VariantGetLong function, 11-9, 11-128
CA_VariantGetLongPtr function, 11-129
CA_VariantGetObjHandle function, 11-130
CA_VariantGetSafeArray function, 11-131 to 11-132
CA_VariantGetSafeArrayPtr function, 11-133 to 11-134
CA_VariantGetShort function, 11-135
CA_VariantGetShortPtr function, 11-136
CA_VariantGetType function, 11-8, 11-137
CA_VariantGetUChar function, 11-138
CA_VariantGetUChartPtr function, 11-139
CA_VariantGetVariantPtr function, 11-140
CA_VariantHasArray function, 11-141
CA_VariantHasBool function, 11-142
CA_VariantHasBSTR function, 11-143
CA_VariantHasCString function, 11-144
CA_VariantHasCurrency function, 11-145
CA_VariantHasDate function, 11-146
CA_VariantHasDispatch function, 11-147
CA_VariantHasDouble function, 11-148
CA_VariantHasError function, 11-149
CA_VariantHasFloat function, 11-150
CA_VariantHasInt function, 11-151
CA_VariantHasIUnknown function, 11-152
CA_VariantHasLong function, 11-9, 11-153
CA_VariantHasNull function, 11-154
CA_VariantHasObjectHandle function, 11-155
CA_VariantHasPtr function, 11-156
CA_VariantHasShort function, 11-9, 11-157
CA_VariantHasUChar function, 11-158
CA_VariantInt function, 11-159
CA_VariantIsEmpty function, 11-160
CA_VariantIUnknown function, 11-161
CA_VariantLong function, 11-162
CA_VariantNULL function, 11-163
CA_VariantSet1DArray function, 11-164 to 11-165
CA_VariantSet2DArray function, 11-166 to 11-167
CA_VariantSetBool function, 11-168
CA_VariantSetBoolPtr function, 11-169

CA_VariantSetBSTR function, 11-170
CA_VariantSetBSTRPtr function, 11-171
CA_VariantSetCString function, 11-172
CA_VariantSetCurrency function, 11-173
CA_VariantSetCurrencyPtr function, 11-174
CA_VariantSetDate function, 11-175
CA_VariantSetDatePtr function, 11-176
CA_VariantSetDispatch function, 11-177
CA_VariantSetDispatchPtr function, 11-178
CA_VariantSetDouble function, 11-179
CA_VariantSetDoublePtr function, 11-180
CA_VariantSetEmpty function, 11-181
CA_VariantSetError function, 11-182
CA_VariantSetErrorPtr function, 11-183
CA_VariantSetFloat function, 11-184
CA_VariantSetFloatPtr function, 11-185
CA_VariantSetInt function, 11-186
CA_VariantSetIntPtr function, 11-187
CA_VariantSetIUnknown function, 11-188
CA_VariantSetIUnknownPtr function, 11-189
CA_VariantSetLong function, 11-190
CA_VariantSetLongPtr function, 11-191
CA_VariantSetNULL function, 11-192
CA_VariantSetSafeArray function, 11-193
CA_VariantSetSafeArrayPtr function, 11-194
CA_VariantSetShort function, 11-195
CA_VariantSetShortPtr function, 11-196
CA_VariantSetUChar function, 11-197
CA_VariantSetUCharPtr function, 11-198
CA_VariantSetVariantPtr function, 11-199
CA_VariantShort function, 11-200
CA_VariantUChar function, 11-201
cables. *See* RS-232 cables.
callback functions
    DDE Library functions, 6-2 to 6-4
        DDE transaction types (table), 6-4
        example using Excel, 6-5 to 6-6
        parameter prototypes (table), 6-3
    GPIB/GPIB-488.2 Libraries
        function tree, 4-3
        ibInstallCallback, 4-10, 4-14 to 4-16
        ibNotify, 4-10, 4-18 to 4-21
        Windows 95/NT
            asynchronous callbacks, 4-10
            driver version
                requirements, 4-10
            ibInstallCallback, 4-14 to 4-16
            ibNotify function, 4-18 to 4-21
            synchronous callbacks, 4-10
    RS-232 Library
        function tree, 5-2
        InstallComCallback, 5-33 to 5-36
    TCP Library functions
        overview, 7-3
        TCP transaction types (table), 7-3
    X Property Library functions
        InstallXPropertyCallback, 9-4, 9-30 to 9-32
        overview, 9-4
        UninstallXPropertyCallback, 9-4, 9-40
character processing, ANSI C, 1-6
CheckForDuplicateAppInstance function, 8-8 to 8-9
classes, ANSI C Library, 1-1 to 1-2
clear functions, GPIB-488.2 Library, 4-3
Clear1D function, 3-8
ClientDDEExecute function, 6-12
ClientDDERead function, 6-13 to 6-14
ClientDDEWrite function, 6-15 to 6-16
clients and servers
    DDE Library functions, 6-2
    TCP Library functions, 7-1
ClientTCPRead function, 7-5
ClientTCPWrite function, 7-6
close functions
    GPIB and GPIB-488.2 Libraries, 4-2
    RS-232 Library, 5-1

CloseCom function, 5-10
CloseCVIRTE function, 8-10
CloseDev function, 4-6, 4-12
CloseFile function, 2-8
CloseInstrDevs function, 4-13
Cls function, 8-11
ComBreak function, 5-11
ComFromFile function, 5-3, 5-12 to 5-13
communications functions. *See* RS-232 Library functions.
CompareBytes function, 2-9 to 2-10
CompareStrings function, 2-11 to 2-12
complex operation functions
    CxAdd, 3-10
    CxAdd1D, 3-11
    CxDiv, 3-12
    CxDiv1D, 3-13
    CxLinEv1D, 3-14 to 3-15
    CxMul, 3-16
    CxMul1D, 3-17
    CxRecip, 3-18
    CxSub, 3-19
    CxSub1D, 3-20
    ToPolar, 3-46
    ToPolar1D, 3-47
    ToRect, 3-48
    ToRect1D, 3-49
ComRd function, 5-14 to 5-15
ComRdByte function, 5-16
ComRdTerm function, 5-17 to 5-18
ComSetEscape function, 5-19 to 5-20
ComToFile function, 5-3, 5-21 to 5-22
ComWrt function, 5-2 to 35-24
ComWrtByte function, 5-25
configuration functions, GPIB Library, 4-2 to 4-3
ConnectToDDEServer function, 6-2, 6-17 to 6-19
ConnectToTCPServer function, 7-7 to 7-8
ConnectToXDisplay function, 9-3, 9-6 to 9-7
ContinuousPulseGenConfig, 10-32 to 10-34
control functions
    ANSI C library, 1-9 to 1-11
    error codes (table), 1-10 to 1-11
    RS-232 Library
        ComBreak, 5-11
        ComSetEscape, 5-19 to 5-20
        FlushInQ, 5-26
        SetComTime, 5-43
        SetCTSMode, 5-7, 5-44 to 5-45
        SetXMode, 5-7, 5-46
Copy1D function, 3-9
CopyBytes function, 2-13
CopyFile function, 8-12 to 8-13
CopyString function, 2-14
Count control, GPIB, 4-6
Count Variables (ibcnt, ibcntl), 4-6, 4-9
CounterEventOrTimeConfig function, 10-35 to 10-38
CounterMeasureFrequency function, 10-39 to 10-42
CounterRead function, 10-32 to 10-33
CounterStart function, 10-44
CounterStop function, 10-45
counter/timer functions, Easy I/O for DAQ Library
    ContinuousPulseGenConfig, 10-32 to 10-34
    CounterEventOrTimeConfig, 10-35 to 10-38
    CounterMeasureFrequency, 10-39 to 10-42
    CounterRead, 10-43
    CounterStart, 10-44
    CounterStop, 10-45
    DelayedPulseGenConfig, 10-46 to 10-48
    FrequencyDividerConfig, 10-49 to 10-52
    ICounterControl, 10-61 to 10-62
    PulseWidthOrPeriodMeasConfig, 10-64 to 10-66
    valid counters (table), 10-7

CreateXProperty function, 9-3, 9-8 to 9-9
CreateXPropType function, 9-3, 9-10 to 9-12
customer communication, *xxvii*, A-1 to A-2
CVILowLevelSupportDriverLoaded function, 8-14 to 8-15
CVIRTEHasBeenDetached function, 8-16 to 8-17
CVIXDisplay global variable, 9-3
CVIXHiddenWindow global variable, 9-4
CVIXRootWindow variable, 9-3
CxAdd function, 3-10
CxAdd1D function, 3-11
CxDiv function, 3-12
CxDiv1D function, 3-13
CxLinEv1D function, 3-14 to 3-15
CxMul function, 3-16
CxMul1D function, 3-17
CxRecip function, 3-18
CxSub function, 3-19
CxSub1D function, 3-20

D

data acquisition functions. *See* Easy I/O for DAQ Library.
data formatting functions. *See* formatting functions; scanning functions; status functions.
DateStr function, 8-18
date/time functions
- ANSI C Library, 1-7 to 1-9
- CA_VariantDate, 11-84
- CA_VariantGetDate, 11-114
- CA_VariantGetDatePtr, 11-115
- CA_VariantHasDate, 11-146
- CA_VariantSetDate, 11-175
- CA_VariantSetDatePtr, 11-176
- configuring DST rules string, 1-8
- DateStr, 8-18
- GetSystemDate, 8-67
- GetSystemTime, 8-68
- modifying DST rules string, 1-8 to 1-9
- SetSystemDate, 8-129
- SetSystemTime, 8-130
- starting year in daylight savings time, 1-9
- suppressing daylight savings time, 1-9
- TimeStr, 8-139

daylight savings time. *See* date/time functions.
DCE device, 5-5
DDE Library functions
- callback function, 6-2 to 6-4
  - functions capable of trigger callback function (table), 6-4
  - parameter prototypes (table), 6-3
- clients and servers, 6-2
- connecting to DDE server, 6-2
- DDE data links, 6-4 to 6-5
- DDE transaction types (table), 6-4
- error conditions, 6-31 to 6-32
- function reference
  - AdviseDDEDataReady, 6-7 to 6-9
  - BroadcastDDEDataReady, 6-10 to 6-11
  - ClientDDEExecute, 6-12
  - ClientDDERead, 6-13 to 6-14
  - ClientDDEWrite, 6-15 to 6-16
  - ConnectToDDEServer, 6-2, 6-17 to 6-19
  - DisconnectFromDDEServer, 6-20
  - GetDDEErrorString, 6-21
  - RegisterDDEServer, 6-2, 6-22 to 6-24
  - ServerDDEWrite, 6-25 to 6-26
  - SetUpDDEHotLink, 6-4, 6-27
  - SetUpDDEWarmLink, 6-4, 6-28
  - TerminateDDELink, 6-29
  - UnregisterDDEServer, 6-30
- function tree (table), 6-1 to 6-2
- Microsoft Excel example, 6-5 to 6-6
- multithreading under Windows 95/NT, 6-6

Delay function, 8-19
DelayedPulseGenConfig function, 10-46 to 10-48
DeleteDir function, 8-20
DeleteFile function, 8-21
DestroyXProperty function, 9-13
DestroyXPropType function, 9-14
Determinant function, 3-21
device control functions, GPIB Library, 4-3, 4-6 to 4-7
device drivers, GPIB, 4-6
device I/O functions, GPIB-488.2 Library, 4-3
device numbers, Easy I/O for DAQ Library, 10-4
digital input/output functions, Easy I/O for DAQ Library
    ReadFromDigitalLine, 10-67 to 10-68
    ReadFromDigitalPort, 10-69 to 10-70
    WriteToDigitalLine, 10-72 to 10-73
    WriteToDigitalPort, 10-74 to 10-75
directory utility functions
    DeleteDir, 8-20
    GetDir, 8-35
    GetDrive, 8-36
    GetFullPathFromProject, 8-51 to 8-52
    GetModuleDir, 8-56 to 8-57
    GetProjectDir, 8-60 to 8-61
    MakeDir, 8-90
    MakePathname, 8-91
    SetDir, 8-113
    SetDrive, 8-114
    SplitPath, 8-131 to 8-132
DisableBreakOnLibraryErrors function, 8-22
DisableInterrupts function, 8-23
DisableTaskSwitching function, 8-24 to 8-26
DisconnectFromDDEServer function, 6-20
DisconnectFromTCPServer function, 7-9
DisconnectFromXDisplay function, 9-15
DisconnectTCPClient function, 7-10
Div1D function, 3-22
Div2D function, 3-23
documentation
    conventions used in manual, *xxv-xxvi*
    LabWindows/CVI documentation set, *xxvi*
    organization of manual, *xviii-xxv*
    related documentation, *xxvi*-xxvii
DotProduct function, 3-24
DSetUpDDEWarmLink function, 6-4 to 6-5
DTE device, 5-5
Dynamic Data Exchange (DDE). *See* DDE Library functions.
dynamic link library, GPIB, 4-5

E

Easy I/O for DAQ Library
    advantages, 10-1 to 10-2
    calls to Data Acquisition Library (note), 10-1
    channel string
        analog input functions, 10-4 to 10-5
        analog output functions, 10-6 to 10-7
    classes, 10-3
    command strings, 10-5 to 10-6
    device numbers, 10-4
    error conditions (table), 10-76 to 10-91
    function reference
        AIAcquireTriggeredWaveforms, 10-8 to 10-13
        AIAcquireWaveforms, 10-14 to 10-16
        AICheckAcquisition, 10-17
        AIClearAcquisition, 10-18
        AIReadAcquisition, 10-19 to 10-20
        AISampleChannel, 10-21 to 10-22
        AISampleChannels, 10-23 to 10-24
        AIStartAcquisition, 10-25 to 10-26
        AOClearWaveforms, 10-27
        AOGenerateWaveforms, 10-28 to 10-29
        AOUpdateChannel, 10-30

AOUpdateChannels, 10-31
ContinuousPulseGenConfig, 10-32 to 10-34
CounterEventOrTimeConfig, 10-35 to 10-38
CounterMeasureFrequency, 10-39 to 10-42
CounterRead, 10-43
CounterStart, 10-44
CounterStop, 10-45
DelayedPulseGenConfig, 10-46 to 10-48
FrequencyDividerConfig, 10-49 to 10-52
GetAILimitsOfChannel, 10-53 to 10-54
GetChannelIndices, 10-55 to 10-56
GetChannelNameFromIndex, 10-57
GetDAQErrorString, 10-58
GetNumChannels, 10-59
GroupByChannel, 10-60
ICounterControl, 10-61 to 10-62
PlotLastAIWaveformsPopup, 10-63
PulseWidthOrPeriodMeasConfig, 10-64 to 10-66
ReadFromDigitalLine, 10-67 to 10-68
ReadFromDigitalPort, 10-69 to 10-70
SetEasyIOMultitaskingMode, 10-71
WriteToDigitalLine, 10-72 to 10-73
WriteToDigitalPort, 10-74 to 10-75
function tree, 10-2 to 10-3
limitations, 10-2
overview, 10-1
valid counters for counter/timer functions (table), 10-7
electronic support services, A-1 to A-2
e-mail support, A-2
EnableBreakOnLibraryErrors function, 8-27
EnableInterrupts function, 8-28
EnableTaskSwitching function, 8-29
END message, GPIB, 4-8
end-of-string (EOS) character, GPIB, 4-8
end-or-identify (EOI) signal, GPIB, 4-8
errno global variable, set by file I/O functions, 1-7
error codes
control functions (table), 1-10 to 1-11
X Property Library, 9-4
error conditions
ActiveX Automation Library, 11-202 to 11-203
Analysis Library functions, 3-51
DDE Library functions, 6-31 to 6-32
Easy I/O for DAQ Library, 10-76 to 10-91
RS-232 Library functions, 5-52 to 5-54
TCP Library functions, 7-25
Error control, GPIB, 4-6
Error (iberr) global variable, 4-6, 4-9
error reporting
Analysis Library functions, 3-4
RS-232 Library functions, 5-3
error-related functions. *See also* status functions.
CA_DisplayErrorInfo, 11-26
CA_GetAutomationErrorString, 11-33
DisableBreakOnLibraryErrors, 8-22
EnableBreakOnLibraryErrors, 8-27
GetAnalysisErrorString, 3-25
GetBreakOnLibraryErrors, 8-31
GetBreakOnProtectionErrors, 8-32
GetDDEErrorString, 6-21
GetFmtErrNdx, 2-26
GetRS232ErrorString, 5-32
GetTCPErrorString, 7-12
GetTCPSystemErrorString, 7-17
GetXPropErrorString, 9-16
ReturnRS232Err, 5-42
SetBreakOnLibraryErrors, 8-109 to 8-110
SetBreakOnProtectionErrors, 8-111 to 8-112

example programs. *See* formatting function programming examples; scanning function programming examples.
ExecutableHasTerminated function, 8-30
executables, launching. *See* standalone executables, launching.
extended character sets, 1-3
external module utility functions
GetExternalModuleAddr, 8-37 to 8-39
LoadExternalModule, 8-83 to 8-87
LoadExternalModuleEx, 8-88 to 8-89
ReleaseExternalModule, 8-101 to 8-102
RunExternalModule, 8-107 to 8-108
UnloadExternalModule, 8-141

F

fax and telephone support numbers, A-2
fdopen function, ANSI C Library, 1-12
file I/O functions
CloseFile, 2-8
errno global variable, 1-7
GetFileInfo, 2-25
OpenFile, 2-30 to 2-31
ReadFile, 2-32 to 2-33
SetFilePtr, 2-42 to 2-43
WriteFile, 2-47 to 2-48
file utility functions
CopyFile, 8-12 to 8-13
DeleteFile, 8-21
GetFileAttrs, 8-40 to 8-41
GetFileDate, 8-42 to 8-43
GetFileSize, 8-44 to 8-45
GetFileTime, 8-46 to 8-47
GetFirstFile, 8-48 to 8-50
GetNextFile, 8-58
RenameFile, 8-103 to 8-104
SetFileAttrs, 8-115 to 8-116
SetFileDate, 8-117
SetFileTime, 8-118 to 8-119
SplitPath, 8-131 to 8-132
FileToArray function, 2-15 to 2-17
FillBytes function, 2-18
FindPattern function, 2-19 to 2-20
floating-point modifiers (%f)
formatting functions, 2-57 to 2-58
scanning functions, 2-69 to 2-71
FlushInQ function, 5-26
FlushOutQ function, 5-27
Fmt, FmtFile, and FmtOut functions. *See* formatting function programming examples; formatting functions.
format codes
formatting functions, 2-53 to 2-54
scanning functions, 2-64 to 2-66
format string
formatting functions, 2-51 to 2-54
examples, 2-52
form of, 2-52
format codes, 2-53 to 2-54
literals, 2-61
scanning functions, 2-62 to 2-66
examples, 2-63
form of, 2-62
format codes, 2-64 to 2-66
literals, 2-75
Formatting and I/O Library functions
function panels
classes and subclasses, 2-3
function tree (table), 2-2
function reference
ArrayToFile, 2-5 to 2-7
CloseFile, 2-8
CompareBytes, 2-9 to 2-10
CompareStrings, 2-11 to 2-12
CopyBytes, 2-13
CopyString, 2-14
FileToArray, 2-15 to 2-17
FillBytes, 2-18
FindPattern, 2-19 to 2-20
Fmt, 2-21 to 2-22, 2-51
FmtFile, 2-23, 2-51

FmtOut, 2-24, 2-51
GetFileInfo, 2-25
GetFmtErrNdx, 2-26
GetFmtIOError, 2-27
GetFmtIOErrorString, 2-28
NumFmtdBytes, 2-29
OpenFile, 2-30 to 2-31
ReadFile, 2-32 to 2-33
ReadLine, 2-34 to 2-35
Scan, 2-36 to 2-37, 2-62
ScanFile, 2-38 to 2-39, 2-62
ScanIn, 2-40 to 2-41, 2-62
SetFilePtr, 2-42 to 2-43
StringLength, 2-44
StringLowerCase, 2-45
StringUpperCase, 2-46
WriteFile, 2-47 to 2-48
WriteLine, 2-49
formatting function programming examples
appending to a string, 2-83
concatenating two strings, 2-82
creating array of file names, 2-84
integer and real to string with literals, 2-79
integer array to binary file, assuming fixed number of elements, 2-80
integer to string, 2-76 to 2-77
list of examples, 2-84
real array to ASCII file in columns with comma separators, 2-79 to 2-80
real array to binary file
assuming fixed number of elements, 2-80
assuming variable number of elements, 2-81
real to string
in floating-point notation, 2-78
in scientific notation, 2-78 to 2-79
short integer to string, 2-77
two integers to ASCII file with error-checking, 2-79
variable portion of real array to binary file, 2-81
writing line containing integer with literals to standard output, 2-84
writing to standard output without linefeed/carriage return, 2-84 to 2-85
formatting functions. *See also* scanning functions; string manipulation functions.
asterisks (*) instead of constants in format specifiers, 2-61
Fmt
description, 2-21 to 2-22
examples, 2-51
FmtFile
description, 2-23
examples, 2-51
FmtOut
description, 2-24
examples, 2-51
format string, 2-51 to 2-54
introductory examples, 2-50 to 2-51
literals in format string, 2-61
purpose and use, 2-50
special nature of, 2-3 to 2-4
formatting modifiers, 2-54 to 2-60. *See also* scanning modifiers.
floating-point modifiers (%f), 2-57 to 2-58
integer modifiers (%i, %d, %x, %o, %c), 2-55 to 2-57
string modifiers (%s), 2-59 to 2-60
freeing resources. *See* resource-freeing functions.
FrequencyDividerConfig function, 10-49 to 10-52
FTP support, A-1

G

gender changer, 5-6
GetAILimitsOfChannel function, 10-53 to 10-54
GetAnalysisErrorString function, 3-25
GetBreakOnLibraryErrors function, 8-31
GetBreakOnProtectionErrors function, 8-32
GetChannelIndices function, 10-55 to 10-56
GetChannelNameFromIndex function, 10-57
GetComStat function, 5-28 to 5-29
GetCurrentPlatform function, 8-33
GetCVIVersion function, 8-34
GetDAQErrorString function, 10-58
GetDDEErrorString function, 6-21
GetDir function, 8-35
GetDrive function, 8-36
GetExternalModuleAddr function, 8-37 to 8-39
GetFileAttrs function, 8-40 to 8-41
GetFileDate function, 8-42 to 8-43
GetFileInfo function, 2-25
GetFileSize function, 8-44 to 8-45
GetFileTime function, 8-46 to 8-47
GetFirstFile function, 8-48 to 8-50
GetFmtErrNdx function, 2-26
GetFmtIOError function, 2-27
GetFmtIOErrorString function, 2-28
GetFullPathFromProject function, 8-51 to 8-52
GetHostTCPSocketHandle function, 7-11
GetInQLen function, 5-30
GetInterruptState function, 8-53
GetKey function, 8-54 to 8-55
GetModuleDir function, 8-56 to 8-57
GetNextFile function, 8-58
GetNumChannels function, 10-59
GetOutQLen function, 5-4, 5-31
GetPersistentVariable function, 8-59
GetProjectDir function, 8-60 to 8-61
GetRS232ErrorString function, 5-32
GetStdioPort function, 8-62
GetStdioWindowOptions function, 8-63
GetStdioWindowPosition function, 8-64
GetStdioWindowSize function, 8-65
GetStdioWindowVisibility function, 8-66
GetSystemDate function, 8-67
GetSystemTime function, 8-68
GetTCPErrorString function, 7-12
GetTCPHostAddr function, 7-13
GetTCPHostName function, 7-14
GetTCPPeerAddr function, 7-15
GetTCPPeerName function, 7-16
GetTCPSystemErrorString function, 7-17
GetWindowDisplaySetting function, 8-69
GetXPropErrorString function, 9-16
GetXPropertyName function, 9-17
GetXPropertyType function, 9-18
GetXPropTypeName function, 9-19
GetXPropTypeSize function, 9-20
GetXPropTypeUnit function, 9-21 to 9-22
GetXWindowPropertyItem function, 9-23 to 9-25
GetXWindowPropertyValue function, 9-26 to 9-29
global variables. *See also* status functions.
    CVIXDisplay, 9-3
    CVIXHiddenWindow, 9-4
    Error (iberr), 4-6, 4-9
    GPIB/GPIB-488.2 libraries, 4-9
    rs232err, 5-3
    Status Word (ibsta), 4-6, 4-9
GPIB and GPIB-488.2 Libraries
    automatic serial polling, 4-7 to 4-8
    board functions, 4-6 to 4-7
    device functions, 4-6 to 4-7
    function panels
        classes and subclasses, 4-4 to 4-5
        function tree (table), 4-2 to 4-4
    function reference
        CloseDev, 4-6, 4-12
        CloseInstrDevs, 4-13

ibInstallCallback, 4-10, 4-14 to 4-16
iblock, 4-17
ibNotify, 4-10, 4-18 to 4-21
ibunlock, 4-22
OpenDev, 4-6, 4-23
ThreadIbcnt, 4-24
ThreadIbcntl, 4-25
ThreadIberr, 4-26 to 4-28
ThreadIbsta, 4-29 to 4-30
writing instrument modules (note), 4-6
global variables, 4-9
GPIB dynamic link library/device driver, 4-5
guidelines and restrictions, 4-6
hardware interrupts and autopolling, 4-8
overview, 4-1
platform and board considerations, 4-9 to 4-10
read and write termination, 4-8 to 4-9
status and error controls, 4-6
timeouts, 4-9
Windows 95/NT, 4-9 to 4-10
multithreading, 4-9
notification of SRQ and other GPIB events, 4-10
writing instrument modules (note), 4-6
GPIB device drivers, 4-5
GPIB.DLL, 4-5
GroupByChannel function, 10-60

H

handshaking for RS-232 communications, 5-6 to 5-8
hardware handshaking, 5-7 to 5-8
software handshaking, 5-7
hardware handshaking, 5-7 to 5-8
hardware interrupts and autopolling, 4-8
help, starting. *See* SystemHelp function.
hidden window for providing X window IDs, 9-3 to 9-4
Histogram function, 3-26 to 3-27

I

ibconf utility, 4-5
ibdev function, 4-6
ibfind function, 4-6
ibInstallCallback function, 4-14 to 4-16
callback function, 4-16
driver version requirements, 4-10
purpose and use, 4-14 to 4-16
SRQI, RQS, and auto serial polling, 4-16
synchronous callbacks, 4-10
iblock function, 4-17
ibNotify function, 4-18 to 4-21
asynchronous callbacks, 4-10
callback function, 4-20 to 4-21
driver version requirements, 4-10
purpose and use, 4-18 to 4-21
rearming error (warning), 4-20
restrictions in asynchronous callbacks, 4-21
SRQI, RQS, and auto serial polling, 4-20
ibunlock function, 4-17
ICounterControl function, 10-61 to 10-62
InitCVIRTE function, 8-70 to 8-71
inp function, 8-72
input/output facilities, ANSI C, 1-6
inpw function, 8-73
InstallComCallback function, 5-33 to 5-36
InstallXPropertyCallback function, 9-4, 9-30 to 9-32
InStandaloneExecutable function, 8-74
integer modifiers (%i, %d, %x, %o, %c)
formatting functions, 2-55 to 2-57
scanning functions, 2-67 to 2-69
interrupts
DisableInterrupts function, 8-23
EnableInterrupts function, 8-28

GetInterruptState function, 8-53
hardware interrupts and autopolling, 4-8
InvMatrix function, 3-28
I/O functions. *See* Easy I/O for DAQ Library; Formatting and I/O Library functions; Standard Input/Output window functions.

K

keyboard utility functions
GetKey, 8-54 to 8-55
KeyHit, 8-75 to 8-76

L

LaunchExecutable function, 8-77 to 8-79
LaunchExecutableEx function, 8-80 to 8-82
launching executables. *See* standalone executables, launching.
LC_COLLATE locale, 1-6
LC_CTYPE locale, 1-5 to 1-6
LC_MONETARY locale, 1-5
LC_NUMERIC locale, 1-5
LC_TIME locale, 1-6
LinEv1D function, 3-29
LinEv2D function, 3-30
literals in format string
formatting functions, 2-61
scanning functions, 2-75
LoadExternalModule function, 8-83 to 8-87
LoadExternalModuleEx function, 8-88 to 8-89
local functions, GPIB-488.2 Library, 4-3
locale, C. *See* C locale.
locale functions, ActiveX Automation Library
CA_GetLocale, 11-35
CA_SetLocale, 11-75 to 11-76
locking functions, GPIB-488.2 Library
iblock, 4-17
ibunlock, 4-22
low-level functions, ActiveX Automation Library
CA_CreateObjectByClassId, 11-18 to 11-19
CA_CreateObjectByProgId, 11-20 to 11-21
CA_CreateObjHandleFromIDispatch, 11-22
CA_GetActiveObjectByClassId, 11-29 to 11-30
CA_GetActiveObjectByProgId, 11-31 to 11-32
CA_GetDispatchFromObjHandle, 11-34
CA_InvokeHelper, 11-36 to 11-40
CA_InvokeHeplerV, 11-41
CA_LoadObjectFromFile, 11-42 to 11-43
CA_LoadObjectFromFileByClassId, 11-44 to 11-45
CA_LoadObjectFromFileByProgId, 11-46 to 11-47
CA_MethodInvoke, 11-48 to 11-49
CA_MethodInvokeV, 11-50
CA_PropertyGet, 11-51 to 11-52
CA_PropertySet, 11-53 to 11-54
CA_PropertySetByRef, 11-55 to 11-56
CA_PropertySetByRefV, 11-57
CA_PropertySetV, 11-58
low-level I/O functions
ANSI C Library, 1-2
GPIB-488.2 Library, 4-3 to 4-4
LPSAFEARRAY typedef, 11-2

M

MakeDir function, 8-90
MakePathname function, 8-91
managing property information. *See* property information, managing.
manual. *See* documentation.
MapPhysicalMemory function, 8-92 to 8-94
mathematical functions, ANSI C, 1-7

matrix algebra functions. *See* vector and matrix algebra functions.
MatrixMul function, 3-31 to 3-32
MaxMin1D function, 3-33
MaxMin2D function, 3-34 to 3-35
Mean function, 3-36
memory access. *See* physical memory access functions.
miscellaneous Easy I/O for DAQ functions
GetAILimitsOfChannel, 10-53 to 10-54
GetChannelIndices, 10-55 to 10-56
GetChannelNameFromIndex, 10-57
GetDAQErrorString, 10-58
GetNumChannels, 10-59
GroupByChannel, 10-60
SetEasyIOMultitaskingMode, 10-71
miscellaneous utility functions
Beep, 8-6
Breakpoint, 8-7
CheckForDuplicateAppInstance, 8-8 to 8-9
CloseCVIRTE, 8-10
Cls, 8-11
CVILowLevelSupportDriverLoaded, 8-14 to 8-15
CVIRTEHasBeenDetached, 8-16 to 8-17
DisableInterrupts, 8-23
EnableInterrupts, 8-28
GetCurrentPlatform, 8-33
GetCVIVersion, 8-34
GetInterruptState, 8-53
GetWindowDisplaySetting, 8-69
InitCVIRTE, 8-70 to 8-71
InStandaloneExecutable, 8-74
RoundRealToNearestInteger, 8-106
SystemHelp, 8-134 to 8-136
TruncateRealNumber, 8-140
Mul1D function, 3-37
Mul2D function, 3-38
multithreading, Windows 95/NT
DDE Library functions, 6-6
GPIB Library functions, 4-9
RS-232 Library functions, 5-8

N

Neg1D function, 3-39
null modem cable, 5-5
NumFmtdBytes function, 2-29

O

one-dimensional array operation functions
Abs1D, 3-5
Add1D, 3-6
Div1D, 3-22
LinEv1D, 3-29
MaxMin1D, 3-33
Mul1D, 3-37
Neg1D, 3-39
Sub1D, 3-43
Subset1D, 3-45
one-dimensional complex operation functions
CxAdd1D, 3-11
CxDiv1D, 3-13
CxLinEv1D, 3-14 to 3-15
CxMul1D, 3-17
CxSub1D, 3-20
ToPolar1D, 3-47
ToRect1D, 3-49
open functions
GPIB Library, 4-2
RS-232 Library, 5-1
OpenCom function, 5-4, 5-37 to 5-38
OpenComConfig function, 5-4, 5-39 to 5-41
OpenDev function, 4-6, 4-23
OpenFile function, 2-30 to 2-31
outp function, 8-95
outpw function, 8-96

P

parallel poll functions,
GPIB-488.2 Library, 4-3
persistent variable functions
GetPersistentVariable, 8-59
SetPersistentVariable, 8-120
physical memory access functions
MapPhysicalMemory, 8-92 to 8-94
ReadFromPhysicalMemory, 8-97 to 8-98
ReadFromPhysicalMemoryEx,
8-99 to 8-100
UnMapPhysicalMemory, 8-142
WriteToPhysicalMemory, 8-143 to 8-144
WriteToPhysicalMemoryEx,
8-145 to 8-146
PlotLastAIWaveformsPopup function, 10-63
port I/O utility functions
inp, 8-72
inpw, 8-73
outp, 8-95
outpw, 8-96
properties. *See also* X Property
Library functions.
definition, 9-2
handles and types, 9-3
property events, handling
GetXPropErrorString, 9-16
InstallXPropertyCallback, 9-4,
9-30 to 9-32
UninstallXPropertyCallback, 9-4, 9-40
property information, managing
CreateXProperty, 9-3, 9-8 to 9-9
DestroyXProperty, 9-13
GetXPropertyName, 9-17
GetXPropertyType, 9-18
property types, managing
CreateXPropType, 9-3, 9-10 to 9-12
DestroyXPropType, 9-14
GetXPropTypeName, 9-19
GetXPropTypeSize, 9-20
GetXPropTypeUnit, 9-21 to 9-22
PulseWidthOrPeriodMeasConfig function,
10-64 to 10-66
PutXWindowPropertyItem function,
9-33 to 9-34
PutXWindowPropertyValue function,
9-35 to 9-37

R

read termination, GPIB, 4-8 to 4-9
ReadFile function, 2-32 to 2-33
ReadFromDigitalLine function,
10-67 to 10-68
ReadFromDigitalPort function, 10-69 to 10-70
ReadFromPhysicalMemory function,
8-97 to 8-98
ReadFromPhysicalMemoryEx function,
8-99 to 8-100
ReadLine function, 2-34 to 2-35
RegisterDDEServer function, 6-2,
6-22 to 6-24
RegisterTCPServer function, 7-2, 7-18
ReleaseExternalModule function,
8-101 to 8-102
remote functions, GPIB-488.2 Library, 4-3
remote hosts
ConnectToXDisplay function, 9-3,
9-6 to 9-7
DisConnectFromXDisplay, 9-15
RemoveXWindowProperty function,
9-38 to 9-39
RenameFile function, 8-103 to 8-104
ResetDevs function no longer supported
(note), 4-11
resource-freeing functions
CA_DiscardObjHandle, 11-25
CA_FreeMemory, 11-27
CA_FreeUnusedServers, 11-28

CA_SafeArrayDestroy, 11-59
CA_VariantClear, 11-8, 11-9, 11-79
RetireExecutableHandle function, 8-105
ReturnRS232Err function, 5-42
RoundRealToNearestInteger function, 8-106
RQS events, and auto serial polling
ibInstallCallback function, 4-16
ibNotify function, 4-20
RS-232 cables, 5-4 to 5-6
DTE to DCE cable configuration (table), 5-5
gender of connectors, 5-6
PC cable configuration (table), 5-4 to 5-5
PC to DTE cable configuration (table), 5-5 to 5-6
RS-232 Library functions
error conditions, 5-52 to 5-54
function panels
classes, 5-2
function tree (table), 5-1 to 5-2
function reference
CloseCom, 5-10
ComBreak, 5-11
ComFromFile, 5-3, 5-12 to 5-13
ComRd, 5-14 to 5-15
ComRdByte, 5-16
ComRdTerm, 5-17 to 5-18
ComSetEscape, 5-19 to 5-20
ComToFile, 5-3, 5-21 to 5-22
ComWrt, 5-2 to 35-24
ComWrtByte, 5-25
FlushInQ, 5-26
FlushOutQ, 5-27
GetComStat, 5-28 to 5-29
GetInQLen, 5-30
GetOutQLen, 5-4, 5-31
GetRS232ErrorString, 5-32
InstallComCallback, 5-33 to 5-36
OpenCom, 5-4, 5-37 to 5-38
OpenComConfig, 5-4, 5-39 to 5-41
ReturnRS232Err, 5-42
SetComTime, 5-43
SetCTSMode, 5-7, 5-44 to 5-45
SetXMode, 5-46
XModemConfig, 5-4, 5-47 to 5-48
XModemReceive, 5-3, 5-4, 5-49 to 5-50
XModemSend, 5-51
handshaking, 5-6 to 5-8
multithreading under Windows 95/NT, 5-8
reporting errors, 5-3
RS-232 cables, 5-4 to 5-6
troubleshooting, 5-3 to 5-4
XModem file transfer functions, 5-3
rs232err global variable, 5-3
RS-485 AT-Serial board, 5-2
RunExternalModule function, 8-107 to 8-108

S

SAFEARRAY data type, 11-2
scanning function programming examples
ASCII file to two integers with error checking, 2-94
ASCII file with comma separated numbers to real array, with number of elements at beginning of file, 2-94
binary file to integer array, assuming fixed number of elements, 2-95
binary file to real array
assuming fixed number of elements, 2-95
assuming variable number of elements, 2-95
integer array containing 1-byte integers to real array, 2-92
integer array to real array, 2-92
with byte swapping, 2-92
list of examples, 2-75 to 2-76
reading integer from standard input, 2-96
reading line from standard input, 2-97

reading string from standard input, 2-96
scanning strings that are not NUL-terminated, 2-91 to 2-92
string containing binary integers to integer array, 2-93
string containing IEEE-format real number to real variable, 2-93
string to integer, 2-85 to 2-86
string to integer and real, 2-87 to 2-88
string to integer and string, 2-89
string to real, 2-86 to 2-87
after finding semicolon in string, 2-90
after finding substring in string, 2-90 to 2-91
skipping over non-numeric characters, 2-89 to 2-90
string to short integer, 2-86
string to string, 2-88
string with comma-separated ASCII numbers to real array, 2-91
scanning functions. *See also* Formatting and I/O Library functions; formatting functions; string manipulation functions.
asterisks (*) instead of constants in format specifiers, 2-74
format string, 2-62 to 2-66
introductory examples, 2-50 to 2-51
literals in format string, 2-75
purpose and use, 2-62
Scan, 2-36 to 2-37, 2-62
ScanFile, 2-38 to 2-39, 2-62
ScanIn, 2-40 to 2-41, 2-62
special nature of, 2-3 to 2-4
scanning modifiers. *See also* formatting modifiers.
floating-point modifiers (%f), 2-69 to 2-71
integer modifiers (%i, %d, %x, %o, %c), 2-67 to 2-69
string modifiers (%s), 2-71 to 2-74
serial communications functions. *See* RS-232 Library functions.
serial poll functions, GPIB-488.2 Library, 4-3
serial polling, automatic. *See* automatic serial polling.
ServerDDEWrite function, 6-25 to 6-26
ServerTCPWrite function, 7-21
Set1D function, 3-40
SetBreakOnLibraryErrors function, 8-109 to 8-110
SetBreakOnProtectionErrors function, 8-111 to 8-112
SetComTime function, 5-43
SetCTSMode function, 5-7, 5-44 to 5-45
SetDir function, 8-113
SetDrive function, 8-114
SetEasyIOMultitaskingMode function, 10-71
SetFileAttrs function, 8-115 to 8-116
SetFileDate function, 8-117
SetFilePtr function, 2-42 to 2-43
SetFileTime function, 8-118 to 8-119
SetPersistentVariable function, 8-120
SetStdioPort function, 8-121 to 8-122
SetStdioWindowOptions function, 8-123 to 8-124
SetStdioWindowPosition function, 8-125 to 8-126
SetStdioWindowSize function, 8-127
SetStdioWindowVisibility function, 8-128
SetSystemDate function, 8-129
SetSystemTime function, 8-130
SetTCPDisconnectMode function, 7-22 to 7-23
SetUpDDEHotLink function, 6-4, 6-27
SetUpDDEWarmLink function, 6-4, 6-28
SetXMode function, 5-7, 5-46
software handshaking, 5-7
Sort function, 3-41
SplitPath function, 8-131 to 8-132

SRQ functions, GPIB-488.2 Library
function tree, 4-3
Windows 95/NT
asynchronous callbacks, 4-10
device version requirements, 4-10
synchronous callbacks, 4-10
SRQI event, and auto serial polling
ibInstallCallback function, 4-16
ibNotify function, 4-20
standalone executables, launching
ExecutableHasTerminated function, 8-30
LaunchExecutableEx function, 8-80 to 8-82
RetireExecutableHandle function, 8-105
TerminateExecutable function, 8-137
Standard Input/Output window functions
GetStdioPort, 8-62
GetStdioWindowOptions, 8-63
GetStdioWindowPosition, 8-64
GetStdioWindowSize, 8-65
GetStdioWindowVisibility, 8-66
SetStdioPort, 8-121 to 8-122
SetStdioWindowOptions, 8-123 to 8-124
SetStdioWindowPosition, 8-125 to 8-126
SetStdioWindowSize, 8-127
SetStdioWindowVisibility, 8-128
standard language additions, ANSI C, 1-3 to 1-6
statistics functions
Histogram, 3-26 to 3-27
Mean, 3-36
Sort, 3-41
StdDev, 3-42
Status control, GPIB, 4-6
status functions. *See also* error-related functions.
Formatting and I/O Library functions
GetFmtErrNdx, 2-26
GetFmtIOError, 2-27
GetFmtIOErrorString, 2-28
NumFmtdBytes, 2-29
RS-232 library
GetComStat, 5-28 to 5-29
GetInQLen, 5-30
GetOutQLen, 5-4, 5-31
GetRS232ErrorString, 5-32
ReturnRS232Err, 5-42
thread-specific, GPIB Library
ThreadIbcnt, 4-24
ThreadIbcntl function, 4-25
ThreadIberr, 4-26 to 4-28
ThreadIbsta, 4-29 to 4-30
Status Word (ibsta) global variable, 4-6, 4-9
StdDev function, 3-42
string manipulation functions
CompareBytes, 2-9 to 2-10
CompareStrings, 2-11 to 2-12
CopyBytes, 2-13
CopyString, 2-14
definition, 2-3
FillBytes, 2-18
FindPattern, 2-19 to 2-20
ReadLine, 2-34 to 2-35
StringLength, 2-44
StringLowerCase, 2-45
StringUpperCase, 2-46
WriteLine, 2-49
string modifiers (%s)
formatting functions, 2-59 to 2-60
scanning functions, 2-71 to 2-74
string processing, ANSI C, 1-6
Sub1D function, 3-43
Sub2D function, 3-44
Subset1D function, 3-45
synchronous callbacks, 4-10
SyncWait function, 8-133
system control functions, GPIB-488.2 Library, 4-3
SystemHelp function, 8-134 to 8-136

T

task switching functions
DisableTaskSwitching, 8-24 to 8-26
EnableTaskSwitching, 8-29
TCP Library functions
callback function, 7-3
clients and servers, 7-1
error conditions, 7-25
function reference
ClientTCPRead, 7-5
ClientTCPWrite, 7-6
ConnectToTCPServer, 7-7 to 7-8
DisconnectFromTCPServer, 7-9
DisconnectTCPClient, 7-10
GetHostTCPSocketHandle, 7-11
GetTCPErrorString, 7-12
GetTCPHostAddr, 7-13
GetTCPHostName, 7-14
GetTCPPeerAddr, 7-15
GetTCPPeerName, 7-16
GetTCPSystemErrorString, 7-17
RegisterTCPServer, 7-2, 7-18
ServerTCPWrite, 7-21
SetTCPDisconnectMode, 7-22 to 7-23
UnregisterTCPServer, 7-24
function tree (table), 7-1 to 7-2
multithreading under Windows 95/NT, 7-4
technical support, A-1 to A-2
telephone and fax support numbers, A-2
TerminateDDELink function, 6-29
TerminateExecutable function, 8-137
thread-specific status functions
function tree, 4-3
ThreadIbcnt, 4-24
ThreadIbcntl function, 4-25
ThreadIberr, 4-26 to 4-28
ThreadIbsta, 4-29 to 4-30
time/date functions. *See* date/time functions.
timeouts, GPIB, 4-9
timer/wait utility functions
Delay, 8-19
SyncWait, 8-133
Timer, 8-138
TimeStr function, 8-139
ToPolar function, 3-46
ToPolar1D function, 3-47
ToRect function, 3-48
Transmission Control Protocol Library functions. *See* TCP Library functions.
Transpose function, 3-50
trigger functions, GPIB-488.2 Library, 4-3
troubleshooting RS-232 Library functions, 5-3 to 5-4
TruncateRealNumber function, 8-140
two-dimensional array operation functions
Add2D, 3-7
Div2D, 3-23
LinEv2D, 3-30
MaxMin2D, 3-34 to 3-35
Mul2D, 3-38
Sub2D, 3-44

U

UninstallXPropertyCallback function, 9-4, 9-40
UnloadExternalModule function, 8-141
UnMapPhysicalMemory function, 8-142
UnregisterDDEServer function, 6-30
UnregisterTCPServer function, 7-24
Utility Library functions
function panels
classes and subclasses, 8-5
function tree (table), 8-1 to 8-4
function reference
Beep, 8-6
Breakpoint, 8-7

CheckForDuplicateAppInstance, 8-8 to 8-9
CloseCVIRTE, 8-10
Cls, 8-11
CopyFile, 8-12 to 8-13
CVILowLevelSupportDriverLoaded, 8-14 to 8-15
CVIRTEHasBeenDetached, 8-16 to 8-17
DateStr, 8-18
Delay, 8-19
DeleteDir, 8-20
DeleteFile, 8-21
DisableBreakOnLibraryErrors, 8-22
DisableInterrupts, 8-23
DisableTaskSwitching, 8-24 to 8-26
EnableBreakOnLibraryErrors, 8-27
EnableInterrupts, 8-28
EnableTaskSwitching, 8-29
ExecutableHasTerminated, 8-30
GetBreakOnLibraryErrors, 8-31
GetBreakOnProtectionErrors, 8-32
GetCurrentPlatform, 8-33
GetCVIVersion, 8-34
GetDir, 8-35
GetDrive, 8-36
GetExternalModuleAddr, 8-37 to 8-39
GetFileAttrs, 8-40 to 8-41
GetFileDate, 8-42 to 8-43
GetFileSize, 8-44 to 8-45
GetFileTime, 8-46 to 8-47
GetFirstFile, 8-48 to 8-50
GetFullPathFromProject, 8-51 to 8-52
GetInterruptState, 8-53
GetKey, 8-54 to 8-55
GetModuleDir, 8-56 to 8-57
GetNextFile, 8-58
GetPersistentVariable, 8-59
GetProjectDir, 8-60 to 8-61
GetStdioPort, 8-62
GetStdioWindowOptions, 8-63
GetStdioWindowPosition, 8-64
GetStdioWindowSize, 8-65
GetStdioWindowVisibility, 8-66
GetSystemDate, 8-67
GetSystemTime, 8-68
GetWindowDisplaySetting, 8-69
InitCVIRTE, 8-70 to 8-71
inp, 8-72
inpw, 8-73
InStandaloneExecutable, 8-74
KeyHit, 8-75 to 8-76
LaunchExecutable, 8-77 to 8-79
LaunchExecutableEx, 8-80 to 8-82
LoadExternalModule, 8-83 to 8-87
LoadExternalModuleEx, 8-88 to 8-89
MakeDir, 8-90
MakePathname, 8-91
MapPhysicalMemory, 8-92 to 8-94
outp, 8-95
outpw, 8-96
ReadFromPhysicalMemory function, 8-97 to 8-98
ReadFromPhysicalMemoryEx, 8-99 to 8-100
ReleaseExternalModule, 8-101 to 8-102
RenameFile, 8-103 to 8-104
RetireExecutableHandle, 8-105
RoundRealToNearestInteger, 8-106
RunExternalModule, 8-107 to 8-108
SetBreakOnLibraryErrors, 8-109 to 8-110
SetBreakOnProtectionErrors, 8-111 to 8-112
SetDir, 8-113
SetDrive, 8-114
SetFileAttrs, 8-115 to 8-116
SetFileDate, 8-117

SetFileTime, 8-118 to 8-119
SetPersistentVariable, 8-120
SetStdioPort, 8-121 to 8-122
SetStdioWindowOptions, 8-123 to 8-124
SetStdioWindowPosition, 8-125 to 8-126
SetStdioWindowSize, 8-127
SetStdioWindowVisibility, 8-128
SetSystemDate, 8-129
SetSystemTime, 8-130
SplitPath, 8-131 to 8-132
SyncWait, 8-133
SystemHelp, 8-134 to 8-136
TerminateExecutable, 8-137
Timer, 8-138
TimeStr, 8-139
TruncateRealNumber, 8-140
UnloadExternalModule, 8-141
UnMapPhysicalMemory, 8-142
WriteToPhysicalMemory, 8-143 to 8-144
WriteToPhysicalMemoryEx, 8-145 to 8-146

V

va_arg() macro, 1-3
variable argument functions, LabWindows/CVI support of, 1-3
VARIANT data type, ActiveX Automation Library, 11-2
variant parameters. *See also* ActiveX Automation Library.
  input parameters, 11-8
  output parameters, 11-8 to 11-9
  variants marked as empty by retrieval functions, 11-9
variant-related functions
  assigning values to variants
    CA_VariantSet1DArray, 11-164 to 11-165
    CA_VariantSet2DArray, 11-166 to 11-167
    CA_VariantSetBool, 11-168
    CA_VariantSetBoolPtr, 11-169
    CA_VariantSetBSTR, 11-170
    CA_VariantSetBSTRPtr, 11-171
    CA_VariantSetCString, 11-172
    CA_VariantSetCurrency, 11-173
    CA_VariantSetCurrencyPtr, 11-174
    CA_VariantSetDate, 11-175
    CA_VariantSetDatePtr, 11-176
    CA_VariantSetDispatch, 11-177
    CA_VariantSetDispatchPtr, 11-178
    CA_VariantSetDouble, 11-179
    CA_VariantSetDoublePtr, 11-180
    CA_VariantSetEmpty, 11-181
    CA_VariantSetError, 11-182
    CA_VariantSetErrorPtr, 11-183
    CA_VariantSetFloat, 11-184
    CA_VariantSetFloatPtr, 11-185
    CA_VariantSetInt, 11-186
    CA_VariantSetIntPtr, 11-187
    CA_VariantSetIUnknown, 11-188
    CA_VariantSetIUnknownPtr, 11-189
    CA_VariantSetLong, 11-190
    CA_VariantSetLongPtr, 11-191
    CA_VariantSetNULL, 11-192
    CA_VariantSetSafeArray, 11-193
    CA_VariantSetSafeArrayPtr, 11-194
    CA_VariantSetShort, 11-195
    CA_VariantSetShortPtr, 11-196
    CA_VariantSetUChar, 11-197
    CA_VariantSetUCharPtr, 11-198
    CA_VariantSetVariantPtr, 11-199

passing values as variants
CA_DefaultValueVariant, 11-8, 11-24
CA_VariantBool, 11-77
CA_VariantBSTR, 11-78
CA_VariantCurrency, 11-83
CA_VariantDate, 11-84
CA_VariantDispatch, 11-85
CA_VariantDouble, 11-86
CA_VariantEmpty, 11-87
CA_VariantError, 11-88
CA_VariantFloat, 11-89
CA_VariantInt, 11-159
CA_VariantIUnknown, 11-161
CA_VariantLong, 11-162
CA_VariantNULL, 11-163
CA_VariantShort, 11-200
CA_VariantUChar, 11-201
querying type of variant
CA_VariantGetType, 11-8, 11-137
CA_VariantHasArray, 11-141
CA_VariantHasBool, 11-142
CA_VariantHasBSTR, 11-143
CA_VariantHasCString, 11-144
CA_VariantHasCurrency, 11-145
CA_VariantHasDate, 11-146
CA_VariantHasDispatch, 11-147
CA_VariantHasDouble, 11-148
CA_VariantHasError, 11-149
CA_VariantHasFloat, 11-150
CA_VariantHasInt, 11-151
CA_VariantHasIUnknown, 11-152
CA_VariantHasLong, 11-9, 11-153
CA_VariantHasNull, 11-154
CA_VariantHasObjectHandle, 11-155
CA_VariantHasPtr, 11-156
CA_VariantHasShort, 11-9, 11-157
CA_VariantHasUChar, 11-158
CA_VariantIsEmpty, 11-160
retrieving values from variants
CA_VariantConvertToType, 11-9, 11-80 to 11-81
CA_VariantCopy, 11-82
CA_VariantGet1DArray, 11-90 to 11-92
CA_VariantGet1DArrayBuf, 11-93 to 11-95
CA_VariantGet1DArraySize, 11-96
CA_VariantGet2DArray, 11-97 to 11-99
CA_VariantGet2DArrayBuf, 11-100 to 11-102
CA_VariantGet2DArraySize, 11-103
CA_VariantGetArrayNumDims, 11-104
CA_VariantGetBool, 11-105
CA_VariantGetBoolPtr, 11-106
CA_VariantGetBSTR, 11-107
CA_VariantGetBSTRPtr, 11-108
CA_VariantGetCString, 11-109
CA_VariantGetCStringBuf, 11-110
CA_VariantGetCStringLen, 11-111
CA_VariantGetCurrency, 11-112
CA_VariantGetCurrencyPtr, 11-113
CA_VariantGetDate, 11-114
CA_VariantGetDatePtr, 11-115
CA_VariantGetDispatch, 11-116
CA_VariantGetDispatchPtr, 11-117
CA_VariantGetDouble, 11-118
CA_VariantGetDoublePtr, 11-119
CA_VariantGetError, 11-120
CA_VariantGetErrorPtr, 11-121
CA_VariantGetFloat, 11-122
CA_VariantGetFloatPtr, 11-123
CA_VariantGetInt, 11-124
CA_VariantGetIntPtr, 11-125
CA_VariantGetIUnknown, 11-126
CA_VariantGetIUnknownPtr, 11-127
CA_VariantGetLong, 11-9, 11-128

CA_VariantGetLongPtr, 11-129
CA_VariantGetObjHandle, 11-130
CA_VariantGetSafeArray, 11-131 to 11-132
CA_VariantGetSafeArrayPtr, 11-133 to 11-134
CA_VariantGetShort, 11-135
CA_VariantGetShortPtr, 11-136
CA_VariantGetUChar, 11-138
CA_VariantGetUChartPtr, 11-139
CA_VariantGetVariantPtr, 11-140
vector and matrix algebra functions
Determinant, 3-21
DotProduct, 3-24
InvMatrix, 3-28
MatrixMul, 3-31 to 3-32
Transpose, 3-50
void HandlePropertyNotifyEvent function, 9-5
void_InitXPropertyLib function, 9-5

W

wait utility functions. *See* timer/wait utility functions.
window functions, standard input/output. *See* Standard Input/Output window functions.
window properties, accessing
GetXWindowPropertyItem, 9-23 to 9-25
GetXWindowPropertyValue, 9-26 to 9-29
PutXWindowPropertyItem, 9-33 to 9-34
PutXWindowPropertyValue, 9-35 to 9-37
RemoveXWindowProperty, 9-38 to 9-39
Windows 95/NT, 4-9 to 4-10
multithreading, 4-9
DDE Library functions, 6-6
GPIB and GPIB-488.2 Libraries, 4-9
TCP Library functions, 7-4
notification of SRQ and other GPIB events, 4-10
asynchronous callbacks, 4-10
driver version requirements, 4-10
synchronous callbacks, 4-10
write termination, GPIB, 4-8 to 4-9
WriteFile function, 2-47 to 2-48
WriteLine function, 2-49
WriteToDigitalLine function, 10-72 to 10-73
WriteToDigitalPort function, 10-74 to 10-75
WriteToPhysicalMemory function, 8-143 to 8-144
WriteToPhysicalMemoryEx function, 8-145 to 8-146

X

X Property Library functions
callback functions, 9-4
communicating with local applications, 9-3
ConnectToXDisplay function, 9-3
error codes, 9-4
function panels, 9-1 to 9-2
function reference
ConnectToXDisplay, 9-6 to 9-7
CreateXProperty, 9-3, 9-8 to 9-9
CreateXPropType, 9-3, 9-10 to 9-12
DestroyXProperty, 9-13
DestroyXPropType, 9-14
DisConnectFromXDisplay, 9-15
GetXPropErrorString, 9-16
GetXPropertyName, 9-17
GetXPropertyType, 9-18
GetXPropTypeName, 9-19
GetXPropTypeSize, 9-20
GetXPropTypeUnit, 9-21 to 9-22
GetXWindowPropertyItem, 9-23 to 9-25
GetXWindowPropertyValue, 9-26 to 9-29

InstallXPropertyCallback, 9-4, 9-30 to 9-32
PutXWindowPropertyItem, 9-33 to 9-34
PutXWindowPropertyValue, 9-35 to 9-37
RemoveXWindowProperty, 9-38 to 9-39
UninstallXPropertyCallback, 9-4, 9-40
void HandlePropertyNotifyEvent, 9-5
void_InitXPropertyLib, 9-5
function tree (table), 9-1 to 9-2
hidden window, 9-3 to 9-4
overview, 9-1
property handles and types, 9-3
predefined property types (table), 9-3
using outside of LabWindows/CVI, 9-5
X interclient communication, 9-2 to 9-3
XModem file transfer functions
purpose and use, 5-3
XModemConfig, 5-4, 5-47 to 5-48
XModemReceive, 5-3, 5-4, 5-49 to 5-50
XModemSend, 5-3, 5-51