LabWindows®/CVI

Standard Libraries Reference Manual

July 1996 Edition

Part Number 320682C-01

Internet Support

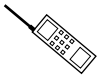
GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabVIEW: lv.support@natinst.com
LabWindows: lw.support@natinst.com
HiQ: hiq.support@natinst.com
VISA: visa.support@natinst.com
Lookout: lookout.support@natinst.com
FTP Site: ftp.natinst.com
Web Address: www.natinst.com

Bulletin Board Support

BBS United States: (512) 794-5422 or (800) 327-3077
BBS United Kingdom: 01635 551422
BBS France: 1 48 65 15 59

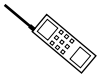
FaxBack Support

(512) 418-1111



Telephone Support (U.S.)

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



International Offices

Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 90 527 2321, France 1 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,
Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 95 800 010 0793,
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, U.K. 01635 523545

National Instruments Corporate Headquarters

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

Warranty

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

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

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

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

Copyright

Trademarks

NI-DAQ®, NI-488.2™, and NI-488.2M™ 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

Chapter 3
Analysis Library 3-1

Chapter 8
Utility Library 8-1

Tables

About This Manual

The *LabWindows/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 use by 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 DOS 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 contains functions that input and output data to files and manipulate the format of data in a program.

- 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 NI-488 and NI-488.2 functions in the LabWindows/CVI GPIB Library, as well as the Device Manager functions in LabWindows/CVI. 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. Detailed descriptions of the NI-488 and NI-488.2 functions can be found in your NI-488.2 function reference manual. 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.

- 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 Microsoft 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 Panels* 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 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:

bold	Bold text denotes a parameter, menu item, return value, function panel item, or dialog box button or option.
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
bold italic	Bold italic text denotes a note, caution, or warning.
`monospace`	Text in this font denotes text or characters that you should literally enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, variables, filenames, and extensions, and for statements and comments taken from program code.
`italic monospace`	Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.
< >	Angle brackets enclose the name of a key. A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys–for example, <Ctrl-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** option from the last dialog box.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files, as in `drivename\dir1name\dir2name\myfile`

IEEE 488, IEEE 488 and IEEE 488.2 refer to the ANSI/IEEE Standard 488.1-1987, IEEE 488.2 and the ANSI/IEEE Standard 488.2-1992, respectively, which define the GPIB.

Abbreviations, acronyms, metric prefixes, mnemonics, symbols, and terms are listed in the *Glossary*.

The LabWindows/CVI Documentation Set

For a detailed discussion of the best way to use the LabWindows/CVI documentation set, see 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 may 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, California: O'Reilly & Associates, 1994. ISBN 0-937175-27-7
- Gettys, James and Robert W. Scheifler. *Xlib—C Language X Interface, MIT X Consortium Standard*. Cambridge, Massachussetts: X Consortium, 1994. ISBN (none)

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 technical support forms for you to complete. These forms are in the appendix, *Customer Communication*, at the end of this manual.

Chapter 1
ANSI C Library

This chapter describes the ANSI C Standard Library as implemented in LabWindows/CVI.

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	<ctype.h>
Character Testing	
Character Case Mapping	
Date and Time	<time.h>
Time Operations	
Time Conversion	
Time Formatting	
Localization	<locale.h>
Mathematics	<math.h>
Trigonometric Functions	
Hyperbolic Functions	
Exp and Log Functions	
Power Functions	
Nonlocal Jumping	<setjmp.h>
Signal Handling	<signal.h>
Input/Output	<stdio.h>
Open/Close	
Read/Write/Flush	
Line Input/Output	
Character Input/Output	
Formatted Input/Output	
Buffer Control	
File Positioning	
File System Operations	
Error Handling	

(continues)

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

General Utilities	<stdlib.h>
String to Arithmetic Expression	
Random Number Generation	
Memory Management	
Searching and Sorting	
Integer Arithmetic	
Multibyte Character Sets	
Program Termination	
Environment	
String Handling	<string.h>
Byte Operations	
String Operations	
String Searching	
Collation Functions	
Miscellaneous	

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. No function panels are provided 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 specified 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, the macro `va_arg (ap, type)` should never be used when `type` is a structure.

Note: ***LabWindows/CVI will 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. The following table 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 *`	`""`	The three-character international currency symbol, plus the character used to separate the international symbol from the monetary quantity.
`currency_symbol`	`char *`	`""`	The 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.
`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 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 `currency_symbol` is separated from non-negative monetary values by a space; else 0.
`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`	The positioning of `positive_sign` for a non-negative monetary quantity, then its `currency_symbol`.
`n_sign_posn`	`char`	`CHAR_MAX`	The 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 items from the `Intl` section of the `WIN.INI` file and appropriate Microsoft Windows functions. Anything not mentioned here has the same behavior 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`.
- `p_cs_precedes` and `n_cs_precedes` are set to 1 if `iCurrency` equals 0 or 2, otherwise they are set to 0.
- `p_sep_by_space` and `n_sep_by_space` are set to 0 if `iCurrency` equals 0 or 1, otherwise they are set to 0.
- `p_sign_posn` and `n_sign_posn` are determined by the value of `iNegCurr` as follows:

Value of iNegCurr	Value of p_sign_posn/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. That is, a German version of LabWindows/CVI would use the German names of months and days.

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 a 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 conversion functions described in these sections 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. The macros are disabled when the LabWindows/CVI debugging level is set to Standard or Extended, so that user protection is available for the arguments to the functions.

String Processing

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

Input/Output Facilities

The function rename fails if the target file already exists. Under Microsoft 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.

The functions fgetpos and ftell set errno to EFILPOS on error.

errno Set by File I/O Functions

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

Under Windows 3.1, errno gives very limited information. If the operating system returns an error, errno is set to EIO.

Under Windows 95 and NT, you can call the Windows SDK GetLastError function to obtain system specific information when errno is set to one of the following values:

```
EACCES
EBADF
EIO
ENOENT
ENOSPC
```

Mathematical Functions

The macro HUGE_VAL defined in the header math.h as well as 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, these macros cannot be used in places where constant expressions are required, such as in global initializations.

Time and Date Functions

Function time returns the number of seconds since January 1, 1990.

Functions mktime and localtime require time zone information to produce correct results. LabWindows/CVI obtains time zone information from the environment variable named 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 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. As an example, the string EST05EDT specifies the time zone information for the eastern part of the United States.

The functions gmtime, localtime, and mktime make corrections for daylight savings time (DST). LabWindows/CVI uses a set of rules for determining when daylight savings time begins and ends. A string in the messages file cvimsgs.txt in the LabWindows/CVI bin directory specifies these rules. The following is the default value of this string.

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

This states that for the years from 1986 to the present, DST begins at 2:00 a.m. on the first Sunday in April, and ends at 2:00 a.m. on the last Sunday in October. For the years from 1967 to 1985, DST begins at 2:00 a.m. on the last Sunday in March, and ends at 2:00 a.m. on the last Sunday in October. You can change the way LabWindows/CVI determines DST by changing this string in the cvimsgs.txt file. The countmsg.exe program must be executed after changing the text file. You should execute the following line.

```
countmsg cvimsgs.txt
```

Control Functions

The assert macro defined by LabWindows/CVI 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 will display a dialog box with a message containing 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 was performing a wait(2V) system call and was waiting until the shell terminated. Callbacks are not called while the command is executing.

Under Windows, the executable can be either an MS DOS or Microsoft Windows executable, including *.exe, *.com, *.bat, and *.pif files. The function does not return until the command terminates, and user keyboard and mouse events are ignored until the command exits. Callbacks for asynchronous events, such as idle events, Windows messages, and 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 would like executed. Refer to your DOS documentation for further help with command.com. DOS executables (.exe, .com, and .bat files) use the settings in _default.pif (in your Windows directory) when they are running. You can change their 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 the function is passed a null pointer, LabWindows/CVI returns a non zero value if a command processor is available. Under UNIX, if the argument is not a null pointer, the program returns a zero. Under Microsoft 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 following error codes.

- -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.
- -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 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. The file must be decompressed before it can be loaded.
- -21 Dynamic-link library (DLL) file was invalid. One of the DLLs required to run this application was corrupt.
- -22 Application requires Microsoft Windows 32-bit extensions.
- -23 Could not find `toolhelp.dll` or `toolhelp.dll` is corrupted.
- -24 Could not allocate a `GetProcUserDefinedHandle`.

The `exit` function does not actually flush and close the open streams. LabWindows/CVI leaves files open so that they may be used from within the Interactive Window after execution of the project terminates. The **Close Libraries** menu option under the **Run** menu performs this library cleanup. This library cleanup is also performed when you restart execution of the project by selecting **Run Project** from the **Run** menu. The argument passed to function `exit` is not used by the LabWindows/CVI environment. Under UNIX, standalone executables created by LabWindows/CVI return the value of the argument passed to the `exit` function.

The UNIX version of LabWindows/CVI works with all the signals supported by UNIX in addition to the ANSI C signals.

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*. Alternatively, you can use LabWindows/CVI function panel help. The following function description is provided because it is an extension of the ANSI C function set.

fdopen

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

Note: ***This function is available only in the Windows version of LabWindows/CVI.***

Purpose

You can use this function to obtain a pointer to a buffered I/O stream from a file handle returned by one of the following functions.

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

You can use the return value just as if you had obtained it from `fopen`.

(Although this function is not in the ANSI standard, it is included in this library because it returns a pointer to a buffered I/O stream.)

Parameters

Input	**fileHandle**	integer	File handle returned by `open` or `sopen`.
	mode	string	Specifies the read/write, binary/text, and append modes.

Return Value

fp	`FILE *`	Pointer to a buffered I/O file stream.

Return Codes

`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.

Chapter 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 reading the other sections of this chapter.

The *Formatting and I/O Library Function Reference* section contains an alphabetical list of function descriptions. This section is helpful for determining 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 these functions are listed in the function reference, 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.

The Formatting and I/O Library Function Panels

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

The first- and second-level bold 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 in plain text are the names of individual function panels. The names of the functions are in bold italics to the right of the function panels. Refer to the *Sample Function Panels for the Formatting and Scanning Functions* section later in this chapter for more information.

Table 2-1. The Formatting and I/O Library Function Tree

Formatting and I/O	
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***

The classes and subclasses in the tree are described below:

- 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 with each panel contains specific information about operating each function panel.

The 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.

The 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 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 instructs the formatting and scanning functions on how to transform the input arguments to the output arguments. For conciseness, format strings are constructed using single-character codes. These codes are

described in detail in the *Using the Formatting and Scanning Functions* section later in this chapter.

You may 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 gives a brief description of each of the functions available in the LabWindows/CVI Formatting and I/O Library. The LabWindows/CVI Formatting and I/O Library functions are arranged alphabetically.

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. The file can later be read back into an array using the `FileToArray` function.

Parameters

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

Return Value

status	integer	Indicates success/failure.

Return Codes

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 **separationStyle** parameter.
-11	Invalid **fieldWidth** parameter.
-12	Invalid **fileAction** parameter.

Parameter Discussion

FileName may be an absolute pathname or a relative file name. If you use a relative file name, the file is created 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 may divide the array data into groups. Groups can be written as either 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–all points of each data group are assumed to be stored consecutively in the data array.

- VAL_DATA_MULTIPLEXED–it is assumed that the first point from each data group is stored together, followed by the second point from each group 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 specify that multiple values be written on each line, **colSepStyle** specifies how the values are separated. 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 have specified 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**. **fileHandle** is the file handle that was returned from the `OpenFile` function and specifies the file to close.

Parameter

Input	**fileHandle**	integer	File handle.

Return Value

status	integer	Result of the close file operation.

Return Codes

`-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** starting at position **buffer#1Index** of **buffer#1** to the **numberofBytes** starting at position **buffer#2Index** of **buffer#2**.

Parameters

Input	**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

result	integer	Result of the compare operation.

Return Codes

-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

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

If **caseSensitive** is zero, alphabetic characters are compared without regard to case. If **caseSensitive** is non-zero, alphabetic characters are considered equal only if they have the same case.

The function returns an integer value indicating 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 NUL-terminated string starting at position **string#1Index** of **string#1** to the NUL-terminated string starting at position **string#2Index** of **string#2**. Both **string#1Index** and **string#2Index** are zero-based.

Parameters

Input	**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

result	integer	Result of the compare operation.

Return Codes

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

The function returns an integer value indicating 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 starting at position **sourceIndex** of **sourceBuffer** to position **targetIndex** of **targetBuffer**.

Parameters

Input	**targetIndex**	integer	Starting position in **targetBuffer**.
	sourceBuffer	string	Source buffer.
	sourceIndex	integer	Starting position in **sourceBuffer**.
	numberofBytes	integer	Number of bytes to copy.
Output	**targetBuffer**	string	Destination buffer.

Return Value

None

Parameter Discussion

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

You can use this function even when **sourceBuffer** and **targetBuffer** overlap.

CopyString

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

Purpose

Copies the string starting at position **sourceIndex** of **sourceString** to position **targetIndex** of **targetString** until an ASCII NUL is copied or **maximum#Bytes** bytes have been copied. Appends an ASCII NUL if no ASCII NUL was copied.

Parameters

Input	**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	**targetString**	string	Destination buffer.

Return Value

None

Parameter Discussion

Both **sourceIndex** and **targetIndex** are zero-based. If you want to use **maximum#Bytes** to prevent from writing beyond the end of **targetString**, make sure that you 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 this function even 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. Can be used with files created using the **ArrayToFile** function. The function handles creating, opening, reading, and closing the file.

Parameters

Input	**fileName**	string	File pathname.
	dataType	integer	Array element data type.
	numberOfElements	integer	Number of elements in array.
	numberOfGroups	integer	Number of Groups in array.
	arrayDataOrder	integer	How groups are ordered in file.
	fileLayout	integer	Direction to write groups in file.
	fileType	integer	ASCII/binary mode.
Output	**array**	void*	Numeric array.

Return Value

status	integer	Indicates success or failure.

Return Code

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

Parameter Discussion

FileName may be an absolute pathname or a relative file name. If you use a relative file name, the file is located 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. Groups can be in the form of either columns or rows. If there are no groups, use 1. This parameter only applies if the file type is ASCII.

If the data is divided into groups, **arrayDataOrder** specifies the order in which the data is to be stored in the array. The two choices are as follows.

- VAL_GROUPS_TOGETHER— all points from one data group are stored together followed by all points from the next data group.
- VAL_DATA_MULTIPLEXED—the first points from each data group are stored consecutively, followed by the second points from each group, etc.

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 starting at position **startingIndex** of **buffer** to the value in the lower byte of **value**. **startingIndex** is zero-based.

Parameters

Input	**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 pattern of bytes is specified by the string **pattern**.

Parameters

Input	**buffer**	string	Buffer to be searched.
	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

ndx	integer	Index in **buffer** where pattern was found.

Return Code

`-1`	Pattern not found.

Parameter Discussion

The buffer searched is the set of **numberofBytes** bytes starting at position **startingIndex** of **buffer**. Exception: If **numberofBytes** is -1, the buffer searched is the set of bytes starting at position **startingIndex** of **buffer** up to the first ASCII NUL. **startingIndex** is zero-based.

If **caseSensitive** is zero, alphabetic characters are compared without regard to case. If **caseSensitive** is non-zero, alphabetic characters are considered equal only if they have the same case. If **startFromRight** is zero, the leftmost occurrence of the pattern in the buffer will be found. If **startFromRight** is non-zero, the rightmost occurrence of the pattern in the buffer will be found.

If the pattern is found, **pattern** returns the index *relative to the beginning of* **buffer** where it found the first byte of the pattern. If the pattern is not found, **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`. The first occurrence is skipped because **startingIndex** is 3. Of the two remaining occurrences, the leftmost is found 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 non-zero:

```
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	**formatString** **source1,…,sourcen**	String. Types must match **formatString** contents.
Output	**target**	Type must match **formatString** contents.

Return Value

n	integer	Number of source format specifiers satisfied.

Return Code

`-1`	Format string error.

Using This Function

This function 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 in error. A complete discussion of this function is in the *Using the Formatting and Scanning Functions* section later in this chapter.

FmtFile

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

Purpose

Formats the **source1 ... sourcen** arguments according to descriptions in the **formatString** argument. The result of the formatting is written into the file corresponding to the **fileHandle** argument, which was obtained by a call to the LabWindows/CVI function `OpenFile`.

Parameters

Input	**fileHandle** **formatString** **source1,…,sourcen**	integer string types must match **formatString** contents	File handle.

Return Value

n	integer	Number of source format specifiers satisfied.

Return Codes

`-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 in error, or `-2` if there was an I/O error. A complete discussion of this function is in the *Using the Formatting and Scanning Functions* section later in this chapter.

FmtOut

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

Purpose

Formats the **source1 ... sourcen** arguments according to descriptions in the **formatString** argument. The result of the formatting is written to the Standard I/O window.

Parameters

Input	**formatString**	String.
	source1,…,sourcen	Types must match **formatString** contents.

Return Value

n	integer	Number of source format specifiers satisfied.

Return Codes

`-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 in error, or `-2` if there was an I/O error. A complete discussion of this function is in the *Using the Formatting and Scanning Functions* section later in this chapter.

GetFileInfo

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

Purpose

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

Parameters

Input	**fileName**	string	Pathname of the file to be checked.
Output	**fileSize**	long	File size or zero.

Return Value

status	integer	Indicates if the file exists.

Return Codes

`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 stating 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

n	integer	Position of error in format string.

Return Code

`-1`	No error.

Using This Function

If the format string of the preceding call contains an error, such as an invalid format, or 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 existed within the string.

Example

```
int i, n;
Scan ("1234", "%s>%d", &i);
n = GetFmtErrNdx ();
/* n will have the value -1, indicating that */
/* there was no error found 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, **GetLastFmtIOError** returns zero (no

error). If the last function that performs I/O encountered an I/O error, **GetLastFmtIOError** returns a nonzero value.

Return Value

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

Return Codes

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	File name is too long.

GetFmtIOErrorString

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

Purpose

Converts the error number returned by **GetLastFmtIOError** into a meaningful error message.

Parameters

Input	**errorNum**	integer	Error Code returned by **GetLastFmtIOErr.**

Return Value

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

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 have been no preceding formatting or scanning calls.

Certain operations using the `FmtFile` and `ScanFile` routines can result in more than 64 KB being formatted or scanned. Because `NumFmtdBytes` returns an integer, its value will not be accurate in these cases. The value returned rolls over when formatting or scanning more than 65,535 bytes.

Example

```
double f;   int n;
Scan ("3.1416", "%s>%f", &f);
n = NumFmtdBytes ();
/* n will have the value 6, indicating that six 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	**fileName**	string	Pathname.
	read/writeMode	integer	Read/write mode.
	action	integer	File pointer reposition location.
	fileType	integer	ASCII/binary mode.

Return Value

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

Return Code

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

Parameter Discussion

fileName is a pathname specifying the file to be opened. If the **read/writeMode** argument is write or read/write, this function creates the file if it does not already exist. If a file is created, it is created with no protection; that is, both reading and writing can be performed on it. Use the function `GetFileInfo` if it is necessary to determine whether a file already exists.

read/writeMode specifies how the file is opened:

- `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 =** write or read/write. After read operations are performed, the file pointer points to the byte following the last byte read. **action** values are as follows:

- `VAL_TRUNCATE` = truncate file (deletes its old contents and positions the file pointer at the beginning of the file.
- `VAL_APPEND` = do not truncate file (all write operations append to end of file).
- `VAL_OPEN_AS_IS` = do not truncate file (positions the file pointer at the beginning of the file.)

fileType specifies whether to treat file as ASCII or binary. When performing I/O on a file in binary mode, no special treatment is given to carriage returns (`CR`) and line feeds (`LF`). 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 `STDIN` into **buffer**. Reading starts at the current position of the file pointer. When the function completes, the file pointer points to the next unread character in the file.

Parameters

Input	**fileHandle** **count**	integer integer	File handle. Number of bytes to read.
Output	**buffer**	string	Input buffer.

Return Value

n	integer	Number of bytes read.

Return Codes

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

Parameter Discussion

fileHandle is the file handle returned by the `OpenFile` function. **fileHandle** points to the file from which you want to read. If **fileHandle** =`0`, input is read from `STDIN`, and no prior `OpenFile` call is needed. **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 number of bytes requested if end of file was reached before byte count was satisfied. Notice that if you open the file in ASCII mode, each `CR LF` combination read is counted as 1 character, because the pair is translated into `LF` when stored in the buffer.

Note: ***This function 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 a linefeed is encountered.

Parameters

Input	**fileHandle** **maximum#Bytes**	integer integer	File handle. Maximum number of bytes to read into line, excluding the ASCII NUL.
Output	**lineBuffer**	string	Input buffer.

Return Value

n	integer	Number of bytes read, excluding linefeed.

Return Codes

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

Parameter Discussion

This function places up to **maximum#Bytes** bytes, excluding the linefeed, into **lineBuffer**. Appends an ASCII NUL to **lineBuffer**. If there are more than **maximum#Bytes** bytes before the linefeed, the extra bytes are discarded.

fileHandle is the file handle that was returned from the `OpenFile` function and specifies the file from which to read the line. The file should be opened in ASCII mode so that a

carriage-return/linefeed combination will be treated as a linefeed. If **fileHandle** is zero, the line will be read from the standard input.

lineBuffer is a character buffer. It 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 the linefeed. Hence, the return value will exceed **maximum#Bytes** if and only if bytes are discarded.

If no bytes are read because the end of the file has been reached, `ReadLine` 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**. The components are then placed into the target parameters.

Parameters

Input	**source** **formatString**	Type must match **formatString** contents string.
Output	**targetptr1,…,targetptrn**	Types must match **formatString** contents.

Return Value

n	integer	Number of target format specifiers satisfied.

Return Code

`-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 in error. A complete discussion of this function is in the *Using the Formatting and Scanning Functions* section later in this chapter.

ScanFile

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

Purpose

Performs the same basic operation as the `Scan` function, except that the source material is obtained from the file referred to by the **fileHandle** argument, which is obtained by calling the LabWindows/CVI function `OpenFile`.

Parameters

Input	**fileHandle** **formatString**	Integer. String.
Output	**targetptr1,…,targetptrn**	Types must match **formatString** contents.

Return Value

n	integer	Number of target format specifiers satisfied.

Return Codes

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

Using This Function

The amount of data read from the file depends on the amount needed to fulfill the formats in the format string. The return value indicates how many target format specifiers were satisfied, `-1` if the format string is in error, or `-2` if there was an I/O error. A complete discussion of this function is in the *Using the Formatting and Scanning Functions* section later in this chapter.

ScanIn

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

Purpose

Performs the same basic operation as the `ScanFile` function, except that the source material is obtained from `STDIN`.

Parameters

Input	**formatString**	String.
Output	**targetptr1,…,targetptrn**	Types must match **formatString** contents.

Return Value

n	integer	Number of target format specifiers satisfied.

Return Codes

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

Using This Function

No argument is required for the source item in the case of the `ScanIn` function. The return value indicates how many target format specifiers were satisfied, -1 if the format string is in error, or -2 if there was an I/O error. A complete discussion of this function is in the *Using the Formatting and Scanning Functions* section later in this chapter.

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	**fileHandle**	integer	File handle returned by `OpenFile`.
	offset	long integer	Number of bytes from origin to position of file pointer.
	origin	integer	Position in file from which to base offset.

Return Value

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

Return Code

-1	Error due to 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

This function can also be used 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 pointer will be at the end of the file.

It is possible to 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 indeterminate values. 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 an indeterminate value.

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	**string**	String.

Return Value

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 NUL-terminated **string** to lowercase.

Parameter

Input/Output	**string**	String.

Return Value

None

StringUpperCase

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

Purpose

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

Parameter

Input/Output	**string**	String.

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 `STDOUT`. Writing starts at the current position of the file pointer, and when the function completes, the file pointer is incremented by the number of bytes written.

Parameters

Input	**fileHandle**	integer	File handle.
	buffer	string	Data buffer.
	count	integer	Number of bytes to write.

Return Value

n	integer	Number of bytes written to the file.

Return Code

`-1`	Error.

Parameter Discussion

fileHandle is the file handle that was returned from the `OpenFile` function. If **fileHandle**=`1`, data is written to `STDOUT` and no prior `OpenFile` call is needed.

buffer is the buffer from which to write data.

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

Using This Function

For files opened in ASCII mode, each LF character is replaced with a CR-LF combination in the output. In this case, the return value does not include the CR character written to the output.

An error can indicate a bad file handle, an attempt to access a protected file, an attempt to write to a file opened as `ReadOnly`, or no more space left on disk.

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	**fileHandle**	integer	File handle.
	lineBuffer	string	Data buffer.
	numberofBytes	integer	Number of bytes to write.

Return Value

n	integer	Number of bytes written. including line feed.

Return Code

-1	I/O error.

Parameter Discussion

If **numberofBytes** is -1, only the bytes in **lineBuffer** before the first ASCII NUL are written, followed by a linefeed.

fileHandle is the file handle that was returned from the `OpenFile` function. The file should be opened in ASCII mode so that a carriage return will be written before the linefeed. If **fileHandle** is 1, the line will be written to the `STDOUT`.

Using This Function

`WriteLine` returns the number of bytes written to the file, excluding the linefeed. 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 usages might be to translate between data stored on external files and the internal forms which the program can manipulate, or to reformat a foreign binary representation into one on which the program can operate.

There are three subclasses of data formatting functions 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 the `Scan` function, 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, ..., sourcen);`

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

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

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

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

  The `FmtOut` function formats the `source1, ..., sourcen` arguments according to descriptions in the `formatstring` argument. The function writes the result of the formatting to Standard Out.

Each of these formatting functions return the number of source format specifiers satisfied. If there is an error in the format string, -1 is returned.

The formatting functions are used to 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 the function Fmt, the target is a data item in memory which is passed to the function by reference. For FmtFile, the target is a file whose handle is passed as the first argument. The LabWindows/CVI function OpenFile returns this handle. For the function FmtOut, the target is Standard Out (typically the display), and in this case the target argument present in the other two functions is omitted. Except for these differences, the following descriptions apply to all the formatting functions.

The target parameter for Fmt must be passed by reference (that is, must be a pointer).

Formatting Functions—Format String

Consider the following formatting function:

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

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 the source material is to be combined into the target.

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

```
"%s < RANGE %i"

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

The character < is a visual reminder of the direction of the data transformation (that is, from the sources to the target), and also separates the single target format specifier from the (perhaps multiple) source format specifiers and literals. The target format specifier can be omitted, in which case a %s string format is assumed. If the target format specifier is omitted, the < character can be omitted also, or retained 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 which indicates the nature of the data items being formatted. `modifiers` is an optional bracket-enclosed sequence of codes which 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:

`s` string. As a source or target specifier, this indicates that the corresponding parameter is a character string. As a target specifier (the default if no target specifier is present), this can mean that numeric source parameters become converted into an ASCII form for inclusion in the target string. See the individual numeric formats, such as `%i` and `%f`, for details of these conversions. Arrays of strings are not allowed. For example, `%10s` is not a valid format string.

Note: ***When a target string is filled in, an ASCII NUL is always placed 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 be used in such a conversion (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.

`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 function performs 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 function does *not* perform conversion to ASCII when converting to or from the string format `%s`. The byte is copied *directly* to or from the string.

Formatting Modifiers

`modifiers` are optional codes used 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.

There is a different set of modifiers for each possible format specifier.

Formatting Integer Modifiers (%i, %d, %x, %o, %c)

`bn` **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 B; therefore, simple 4 B integers do not need this modifier. The modifier b2 represents short integers. The modifier b1 represents single-byte integers.

`in` **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 reads the 10 integer values from the third through the twelfth elements of the array. The `i` modifier is valid only if `rep` is present. If you use the `i` modifier with the `z` modifier, then `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 data in the string is treated as an integer array. The `z` modifier is valid only if `rep` is present.

`rn` **Specify Radix.** The `r` integer modifier specifies the radix of the integer argument, which is important if the integer was to be converted into string format. Legal radixes are 8 (octal), 10 (decimal, the default), 16 (hexadecimal), and 256 (a special radix representing single 8-bit ASCII characters).

`wn` **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 integer is converted 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, an asterisk (`*`) will be inserted into the string to signify an overflow. The default for `n` is zero, which indicates that the integer can occupy whatever space is necessary.

`pc` **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 specified 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 integer argument is considered a signed two's complement number. This is the default interpretation of integers, so the `s` modifier is never explicitly required.

`u` **Specify as Unsigned.** The `u` integer modifier indicates that the integer is considered an unsigned integer.

`onnnn` **Specify Byte Ordering.** The `o` integer modifier is used 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).

In a `Fmt` function, the buffer containing the raw instrument data should have the `o` modifier describing the byte ordering. The buffer without the `o` modifier is guaranteed to be in the mode of the host processor. In other words, LabWindows/CVI will reverse the byte ordering of the buffer without the `o` modifier depending on which architecture the program is running on.

For example, if your GPIB instrument sends two-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, the `Fmt` function should appear as follows:

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

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

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

Formatting Floating-Point Modifiers (%f)

`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`. 8 and 4 are the only valid values for *n*.

in **Specify Array Offset.** You use the `i` modifier to specify an offset within a floating-point array argument. It indicates the location within the array where processing is to begin. *n* is the zero-based index of the first element to process. Thus, `%10f[i2]` applied to a source floating-point array reads the 10 floating-point values from the third through the twelfth elements of the array. The `i` modifier is valid only if `rep` is present. If the `i` modifier is used with the `z` modifier, then `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 data in the string is treated as a floating-point array. The `z` modifier is valid only if `rep` is present.

`wn` **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 value is converted to a string format. Any non-negative value can be entered here. If `n` is less than the number of digits required to represent the floating-point number, an asterisk (`*`) will be inserted into the string to signify an overflow. The default for `n` is zero, which indicates that the value can occupy whatever space is necessary.

`pn` **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 the `p`*n* modifier is omitted, the default value is `p6`**.**

`en` **Specify as Scientific Notation.** The `e` floating-point modifier specifies that a value be converted to string format in scientific notation. If omitted, floating-point notation is used. *n* is optional and specifies the number of digits in the exponent. For example, `%f[e2]` formats 10.0 as 1.0e+01. If *n* is omitted, a default of three is used.

`f` **Specify as Floating-Point Notation.** The `f` floating-point modifier specifies the value to be converted 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 floating-point value is truncated instead of rounded. This is the default.

`r` **Round.** The `r` floating-point modifier indicates that in floating-point to integer transformations, the floating-point value is rounded instead of truncated. The default method is truncation.

Note: ***The value can be represented 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.***

Formatting String Modifiers (`%s`)

`in` **Specify Array Offset.** The `i` string modifier specifies an offset within a string. It indicates the location within the string where processing is to begin. *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. The data is appended beginning at the first occurrence of an ASCII NUL in the target string.

`wn` **Specify String Size.** When modifying a source format specifier, the `w` string modifier specifies the maximum number of bytes to be consumed from the string argument. You can enter any non-negative value here, the default being 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, as many bytes are stored 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 the `q` modifier is used, or blanks if the `q` modifier is not present.

When the `w` string modifier is used in conjunction with the `a` string modifier, *n* indicates the number of bytes to append to the string excluding the terminating ASCII NUL.

If `w`*n* modifies a target string and *n* is larger than the number of bytes in the target argument, the target string is overwritten in compiled C.

`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.

`tn` **Terminate on Character.** When applied to a source string, the `t` string modifier specifies that the source string is terminated on the first occurrence of the character *n*, where *n* is the ASCII value of the character. Thus, `%s[t44]` causes reading of the source string to stop on an ASCII comma. Using `%s[t44]` and the source string `Hello, World!` as an example, `Hello` is placed 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 is present, reading of the source string stops on an ASCII NUL. This modifier has no effect when applied 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 number of bytes specified with the `w` modifier have been read.

`t#` **Terminate on Number.** This is equivalent to repeating the `t` modifier with the ASCII values of the characters `+`, `-`, and `0` through `9`. It specifies that reading of the source string be terminated upon occurrence of a numeric expression. Using `%s[t#]` with the source string `ab567`, `ab` is placed in the target.

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

Often, one or more integer values are required in a format specifier. 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. Alternatively, 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.

You can use the asterisk in the following format specifier elements:

`rep`	For integer or floating-point arrays
`in`	For integer or floating-point arrays, or strings
`wn`	For any format specifier
`pn`	For floating-point specifiers only
`en`	For floating-point specifiers only
`rn`	For integer specifiers only

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

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

Fmt, FmtFile, FmtOut—Literals in the Format String

Literal characters appearing 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.

Since 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. You then can use the function `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, they should be preceded by `%`.

Scanning Functions

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

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

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

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

  The `ScanFile` function reads data from the file corresponding to the `handle` argument and applies transformations to it according to descriptions in the `formatstring` argument. The results of the transformations are placed into the `targetptr1 ... targetptrn` arguments.

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

  The `ScanIn` function reads data from standard input and applies transformations to it according to descriptions in the `formatstring` argument. The results of the transformations are placed into the `targetptr1 ... targetptrn` arguments.

All of the above functions return the number of `target` format specifiers satisfied. The function returns a -1 if there is an error in the format string.

The scanning functions break apart a source item into component parts and store the parts into parameters passed to the function. The only difference between the three functions is the location of the source data. For the function `Scan`, the source item is a data item in memory which is passed to the function. For `ScanFile`, the source item is a file, whose handle is passed as the first argument. The handle is obtained by a call to the LabWindows/CVI function `OpenFile`. For the function `ScanIn`, the source is taken from Standard In (typically the keyboard), and the source argument present in the other two functions is omitted.

All target parameters must be passed 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 are of the following 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:

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

The character > is a visual reminder of the direction of the data transformation, and also separates the single source format specifier from the (possibly multiple) target format specifiers and literals. The source format specifier can be omitted, in which case a `%s` string format is assumed. If the source format specifier is omitted, the > character can be omitted also, or retained 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 which indicates the nature of the data items being formatted. `modifiers` is an optional bracket enclosed sequence of codes which further describe the data format.

The following are examples of format specifiers.

```
%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:

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 are consumed depends on the target specifier. If the target specifier is %s, bytes are consumed until a termination character is encountered (see the t modifier for strings for more information on termination characters). If the target specifier is one of the numeric formats, bytes are consumed as long as they correspond to the pattern for the particular numeric item being converted. Leading spaces and tabs are skipped unless the y modifier is used.

Note: ***When a target string is filled in, an ASCII NUL is always placed 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 bytes from the source argument are to be consumed only until a linefeed is encountered. Also, when modified with c as in %l[c], a comma is used as the target string terminator in place of white space characters.

i integer. As a source or target specifier this 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 integer is converted into digits of the specified radix (default is decimal). As a target specifier in conversions from string format, bytes of the source parameter are consumed as long as they match the pattern of integer ASCII numbers in the appropriate radix, or until the end of the string is encountered. The scanned characters are converted to integer values and placed into the corresponding target parameter, which is an integer or integer array passed by reference. If the format is repeated, the operation is repeated the appropriate number of times with successive elements of the integer array parameter.

The pattern for integer ASCII numbers depends on the radix of the number, and 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 indicates a %i format with decimal radix. Since decimal is the default radix for integers, %d is equivalent to %i, and is included for compatibility with the C scanf family of functions.

`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 floating-point value is converted into ASCII form. As a target specifier in conversions from string format, bytes of the source parameter are consumed as long as they match the pattern of floating-point ASCII numbers, or until the end of the string is encountered. The scanned characters are converted to a floating-point value and placed into the corresponding floating-point or floating-point array target parameter. If the format is repeated, the operation is repeated the appropriate number of times with successive elements of the array parameter. The pattern for floating-point ASCII numbers is an optional sign (+ or −), a series of one or more decimal digits possibly containing a decimal point, and an optional exponent consisting 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 a byte of the source parameter is to be consumed, and the scanned character placed directly into the corresponding target parameter, which is an integer passed by reference. If the format is repeated, this operation is repeated the appropriate number of times and the results stored into successive elements of the integer array.

Scanning Modifiers

`modifiers` are optional codes used 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. There is a different set of modifiers for each possible format specifier.

Scanning Integer Modifiers (%i, %d, %x, %o, %c)

`bn` **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 B; therefore, simple 4 B integers do not need this modifier. The modifier b2 represents short integers. The modifier b1 represents single-byte integers.

`in` **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 is to begin. *n* is the zero-based index of the first element to process. Thus, `%10d[i2]` applied to a source integer array reads the 10 integer values from the third through the twelfth elements of the array. The `i` modifier is valid only if `rep` is present. If the `i` modifier is used with the `z` modifier, then `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 data in the string is treated as an integer array. The `z` modifier is valid only if `rep` is present.

`rn` **Specify Radix.** The `r` integer modifier specifies the radix of the integer argument, which is important if the integer is converted from a string format. Legal radixes are 8 (octal), 10 (decimal, the default), 16 (hexadecimal), and 256 (a special radix representing single 8-bit ASCII characters).

`wn` **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 integer is converted 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, an asterisk (`*`) will be inserted into the string to signify an overflow. The default for `n` is zero, which indicates that the integer can occupy whatever room is necessary.

`s` **Specify as Two's Complement.** The `s` integer modifier indicates that the integer argument is to be considered a signed two's complement number. This is the default interpretation of integers, so the `s` modifier is not required.

`u` **Specify as Non-negative.** The `u` integer modifier indicates that the integer is to be considered a non-negative integer.

`x` **Discard Terminator.** The `x` integer causes the character that terminated the numeric data to be discarded. In this way, terminator characters can be skipped when reading lists of numeric input. Thus, `%3i[x]` reads three integer numbers, disregarding the terminator character which 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 there is no target argument to correspond to the target specifier. The data that otherwise is placed in the target argument is discarded instead. The count returned by the `Scan/ScanFile/ScanIn` functions will *include* the target specifier even if the `d` modifier is used.

`onnnn` **Specify Byte Ordering.** The `o` integer modifier is used 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).

In a `Scan` function, the buffer containing the raw instrument data should have the `o` modifier describing the byte ordering. The buffer without the `o` modifier is guaranteed to be in the mode of the host processor. LabWindows/CVI will reverse the byte ordering of the buffer without the `o` modifier depending on which architecture the program is running.

For example, if your GPIB instrument sends two-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 two-byte binary data in Motorola byte order, the `Scan` function should appear as follows.

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

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

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

Scanning Floating-Point Modifiers (%f)

`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 B; therefore, double-precision values do not need this modifier. Single-precision floating-point values are indicated by `b4`. 8 and 4 are the only valid values for *n*.

`in` **Specify Array Offset.** You can use the `i` floating-point modifier to specify an offset within a floating-point array argument. It indicates the location within the array where processing is to begin. *n* is the zero-based index of the first element to process. Thus, `%10f[i2]` applied to a source floating-point array reads the 10 floating-point values from the third through the twelfth elements of the array. The `i` modifier is valid only if `rep` is present. If you use the `i` modifier with the `z` modifier, then `n` is in terms of bytes.

`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 data in the string is treated as a floating-point array. The `z` modifier is valid only if `rep` is present.

`wn` **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 value is converted 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, an asterisk (`*`) will be inserted into the string to signify an overflow. The default for *n* is zero, which indicates that the value can occupy whatever space is necessary.

`pn` **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. Significant digits may be lost in attempting to conform to the precision specification.

If the p*n* modifier is omitted, a default of p6 is used. The p modifier is valid for sources only.

en **Specify as Scientific Notation.** The e floating-point modifier indicates that the string representation of the floating-point value is in scientific notation. If omitted, non-scientific notation is used. *n* is optional and specifies the number of digits to use in the exponent. For example, %f[e2] causes 10.0 to be formatted as 1.0e+01. If *n* is omitted, a default of three is used. 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 in non-scientific notation. This is the default even when the f modifier is not present.

x **Discard Terminator.** The x floating-point modifier causes the character that terminated the numeric data to be discarded. In this way, terminator characters can be skipped when reading lists of numeric input. Thus, %3f[x] reads three floating-point numbers, disregarding the terminator character which appears after each one; this specifier could then be used to scan the string 3.5, 7.6, −32.4.

d **Discard Data.** When applied to a target specifier, the d modifier indicates there is no target argument to correspond to the target specifier. The data that otherwise is placed in the target argument is discarded instead. The count returned by the Scan/ScanFile/ScanIn functions will *include* the target specifier even if the d modifier is used.

Scanning String Modifiers (%s)

in **Specify Array Offset.** The i string modifier specifies an offset within a string. It indicates the location within the string where processing is to begin. *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.

wn **Specify String Size.** When modifying a source format specifier, the w string modifier specifies the maximum number of bytes from the source string to be used for filling the target arguments. You can enter any non-negative value here, the default being zero, which indicates that the entire string can be used. (For ScanFile and ScanIn, the entire source string is consumed 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, as many bytes are stored 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 the q modifier is used or blanks if the q modifier is not present.

When the w modifier is used in conjunction with the a modifier, *n* indicates the number of bytes to append to the string excluding the terminating ASCII NUL.

If wn modifies a target string and *n* is larger than the number of bytes in the target argument, the target argument is overwritten in compiled C.

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.

y **Append with Spacing.** When the source is a string and the y modifier is applied to a target string format specifier, the target string is filled with bytes from the source string without skipping leading spaces or tabs.

t*n* **Terminate on Character.** When applied to a source string, the t modifier specifies that the source string is terminated on the first occurrence of the character *n*, where *n* is the ASCII value of the character. Thus, %s[t44] causes reading of the source string to stop 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 is present, reading of the source string stops on an ASCII NUL.

When applied to a target string that is being filled from a source string, the t modifier specifies that filling of the target is terminated on the first occurrence of the character *n*, where *n* is the ASCII value of the character. Thus, %s[t59] causes reading of the source string to stop on an ASCII semicolon. More than one t modifier can occur in the same specifier, in which case filling of the target terminates when any of the terminators occur. If no t modifier is present, filling of the target stops 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 reading of the source string terminates when all of the targets are full or when the number of bytes specified with the w modifier have been read. When applied to a target string, t− specifies that filling of the target string terminates when the source is exhausted or when the number of bytes specified with the w modifier have been placed into the target.

t# **Terminate on Number.** This is equivalent to repeating the t modifier with the ASCII values of the characters +, −, and 0 through 9. When applied to a source (target), it specifies that reading of the source string (filling of the target string) be terminated upon occurrence of a numeric expression. Using %s>%s[t#]%d with the source string ab567, ab is placed in the first target and the integer 567 is placed in the second target.

`x` **Discard Terminator.** When applied to a target string, the `x` modifier specifies that the terminating character be discarded before the next target is filled in. Using `%s>%s[xt59]%s[xt59]` with the source string `"abc;XYZ;"`, `"abc"` is placed in the first target and `"XYZ"` is placed in the second target.

`d` **Discard Data.** When applied to a target specifier, the `d` modifier indicates that there is no target argument to correspond to the target specifier. The data that otherwise is placed in the target argument is discarded instead. The count returned by the `Scan`/`ScanFile`/`ScanIn` functions will *include* the target specifier even if the `d` modifier is used.

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. Alternatively, 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.
`in`	For integer or floating-point arrays, or strings.
`wn`	For any format specifier.
`pn`	For floating-point specifiers only.
`en`	For floating-point specifiers only.
`rn`	For integer specifiers only.

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

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

Scan, ScanFile, ScanIn—Literals in the Format String

Literal characters appearing in a scanning function format string indicate that the literal characters are expected in the source parameter. They are not stored into any target parameter, but are skipped over when encountered. If a literal character specified in the format string fails to appear in the source in the expected position, the scanning function immediately returns.

Some formats may have been correctly detected in the input, and the corresponding target parameters will have been filled in. Formats situated after the literal which did not appear, however, will not have been executed.

The function return value can be used to determine exactly how many target parameters were actually fulfilled by the input. You can use the function `NumFmtdBytes` to determine 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. The function `GetFmtErrNdx` can then be used 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, they should be preceded by `%`.

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
Long 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
A Variable Portion of a Real Array to a Binary File
Concatenating Two Strings
Appending to a String
Creating an Array of File Names
Writing a Line Containing an Integer with Literals to the Standard Output
Writing to the Standard Output without a Linefeed/Carriage Return

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

String to Integer
String to Long 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 NUL-Terminated
Integer Array to Real Array
Integer Array to Real Array with Byte Swapping
Integer Array Containing 1-Byte Integers to Real Array
String Containing Binary Integers to Integer Array
String Containing 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 Number of Elements at 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, with Number of Elements at Beginning of File
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 the `Fmt`, `FmtFile`, and `FmtOut` functions from the Formatting and I/O Library. To eliminate redundancy, error checking on I/O operations has been omitted from all of 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" */
```

Remarks

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

Long Integer to String

```
char buf[20];
long a;
a = 123456;
Fmt (buf, "%s<%i[b4]", a);       /* result: "123456" */
a = 123456;
Fmt (buf, "%s<%x[b4]", a);       /* result: "1e240" */
a = 123456;
Fmt (buf, "%s<%o[b4]", a);       /* result: "361100" */
a = -1;
Fmt (buf, "%s<%i[b4]", a);       /* result: "-1" */
a = -1;
Fmt (buf, "%s<%i[b4u]", a);         /* result: "4294967295" */
a = 123456;
Fmt (buf, "%s<%i[b4w8]", a);        /* result: "  123456" */
a = 123456;
Fmt (buf, "%s<%i[b4w8p0]", a);      /* result: "00123456" */
a = 123456;
Fmt (buf, "%s<%i[b4w4]", a);        /* result: "*456" */
```

Remarks

The results shown are the contents of `buf` after each call to `Fmt`. The last call demonstrates what occurs when the width specified by the `w` modifier 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" */
```

Remarks

The results shown are the contents of `buf` after each call to `Fmt`. The last two calls demonstrate that very large and very small values are sometimes forced 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: "1.234500e+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*" */
```

Remarks

The results shown are the contents of `buf` after each call to `Fmt`. The last call demonstrates what occurs when the width specified by the `w` modifier 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);
```

Remarks

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);
```

Remarks

`OpenFile` opens the file `FILE.DAT` as an ASCII file for writing only. If the function 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 there are two source specifiers in the format string).

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);
```

Remarks

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, the linefeed is automatically written as a linefeed/carriage return combination.

Note: ***If the format string is*** `"%s[w15]<%f,"`***, the number and the comma are left-justified 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)
```

Remarks

The `FmtFile` call writes all 100 elements of the integer array `readings` to a file in binary form. If the `FmtFile` call is successful, `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);
```

Remarks

The `FmtFile` call writes all 100 elements of the real array `waveform` to a file in binary form. If the `FmtFile` call is successful, `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);
}
```

Remarks

This example shows how a function can be used to write an array of real numbers to a binary file. The function's 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. The two asterisks (`*`) in the format string are matched to `count`. For instance, if `count` is 100, then the format string is equivalent to `%100f<100f`.

A 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)
}
```

Remarks

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

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

Remarks

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);
```

Remarks

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, `StringLength` is used instead of `NumFmtdBytes`, because `NumFmtdBytes` would return only the number of bytes appended.

Creating an Array of File Names

```
char *fname_array[4];
int i;
fname_array[0] = "               ";    /* 13 spaces */
fname_array[1] = "               ";    /* 13 spaces */
fname_array[2] = "               ";    /* 13 spaces */
fname_array[3] = "               ";    /* 13 spaces */
for (i=0; i < 4; i++)
   Fmt (fname_array[i], "%s<FILE%i[w4p0].DAT", i);
```

Remarks

To allocate the space for each filename in the array, a separate constant string must be assigned to each array element. Then `Fmt` is used to format each file name. The resulting file names are `FILE0000.DAT`, `FILE0001.DAT`, `FILE0002.DAT`, and `FILE0003.DAT`.

Writing a Line Containing 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);
```

Remarks

In this example, the output is as follows:

```
A = 12

B = 34
```

Writing to the Standard Output without a Linefeed/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);
```

Remarks

This example demonstrates how to write to the Standard Output without a linefeed/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 the `Scan`, `ScanFile`, and `ScanIn` functions 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 */
```

Remarks

When locating an integer in a string, `Scan` skips over white space characters such as spaces, tabs, linefeeds, and carriage returns. If a non-numeric character other than a white space character, +, or − is found before the first numeric character, the `Scan` call fails. Thus, `Scan` fails on the `x` in `x32`; it leaves the value in `a` unmodified 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 */
```

Remarks

When the `%i` specifier is used, numeric characters are interpreted as decimal, even when they might appear to be octal (as in `032`) or hexadecimal (as in `32a`). When the `%o` specifier is used, the numeric characters (`01234567`) are always interpreted as octal. When the `%x` specifier is used, the numeric characters (`0123456789abcdef`) are always interpreted 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 only the first 3 bytes of the string are scanned.

String to Long Integer

```
char *s;
long a;
int n;
s = "99999";
n = Scan (s, "%s>%i[b4]", &a);   /* result: a = 99999, n = 1 */
s = "303237";
n = Scan (s, "%s>%o[b4]", &a);   /* result: a = 99999, n = 1 */
s = "ffff";
n = Scan (s, "%s>%x[b4]", &a);   /* result: a = 65535, n = 1 */
```

Remarks

Scan extracts long integers from strings in the same way it extracts integers. The only differences are that the b4 modifier must be used and the target argument must be a long integer. See the *String to Integer* example earlier in this section for more details on using Scan to extract integers and long 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 */
```

Remarks

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 a non-numeric character other than a white space character, +, or − is found 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 only the first 4 bytes of the string are scanned.

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 */
```

Remarks

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, the x modifier is used to discard the separating comma.

In the third call, there is a comma separator 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" */
```

Remarks

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 not present, the substring is considered 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" */
```

Remarks

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 specified should be at least one less than the width of the target character buffer.

When the `w` modifier is used and the string extracted is smaller than the width specified, the remaining bytes in the target string are blank-filled. However, if the `q` modifier is also used, 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 */
```

Remarks

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 */
```

Remarks

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 a number is encountered. The `d` modifier indicates that the bytes must be discarded because there is no argument corresponding to the specifier. When the `Scan` call succeeds, it returns 2, indicating that two target specifiers were satisfied, even though there is only one target argument.

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 */
```

Remarks

Some strings returned by programmable instruments contain headers that consist of numeric as well as non-numeric data and are terminated by a particular character, such as a semicolon. This example shows how such a header can be skipped.

The format string contains two target specifiers. In the first specifier (`%s[xdt#]`), the `t#` modifier instructs `Scan` to read bytes from `s` until a number is encountered. The `d` modifier indicates that the bytes must be discarded because there is no argument corresponding 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 there is only one target argument.

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 */
```

Remarks

This example is similar to the previous one, except that portion of the string to be skipped is terminated by a substring (`DATA`) rather than by a single character. The Formatting and I/O Library function `FindPattern` is used to find the index where `DATA` begins in `s`. Four is added to the index so that it points to the first byte after `DATA`. The index is then passed to `Scan` and matched with the asterisk (`*`) in the format string.

In this example, `FindPattern` returns 15, and `index` is 19. When `index` is matched to the asterisk in the format string in the `Scan` call, the format string is interpreted as `%s[i19]>%f`. The `i19` indicates that the first 19 bytes of `s` should be ignored. `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 */
```

Remarks

The `x` modifier causes the comma separators to be discarded.

`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 NUL-Terminated

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

Remarks

All of the previous examples assume that `s` is a NUL-terminated string. However, when reading data from programmable instruments using the GPIB and RS-232 Library functions, the data transferred is not NUL-terminated. This example uses `ibrd` to read up to 15 B 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 NUL-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 NUL-terminated.

Integer Array to Real Array

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

Remarks

Each integer in `ivals` is converted to real number and then written into `dvals`.

Integer Array to Real Array with Byte Swapping

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

Remarks

Each integer in `ivals` is byte-swapped, converted to a real number, and written 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 this data is read into an integer array, the placement of the bytes is such that the high byte is interpreted as the low byte. The `o10` modifier specifies that the bytes be interpreted in the opposite order.

Integer Array Containing 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);
```

Remarks

Sometimes, each element in an integer array is used 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 one 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. This causes `Scan` to treat the 1-byte integers as unsigned values (from 0 to 255).

String Containing Binary Integers to Integer Array

```
char s[200];   /* string containing 100 2-byte integers */
int ivals[100];/* 100 2-byte integers */
Scan (s, "%100i[z]>%100i", ivals);
Scan (s, "%97i[zi6]>%97i", ivals);
```

Remarks

Sometimes data from a programmable instrument is read 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 is used to indicate that the source is actually a character buffer.

In some cases, the integer data may 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 is used to indicate that the first 6 bytes of `s` are to be ignored.

Note: ***When the*** `i` ***modifier is used in conjunction with a character buffer, the number following 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 the*** `i` ***modifier is used in conjunction with an array variable, the number following the*** `i` ***indicates the number of array elements to ignore.***

String Containing 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);
```

Remarks

This example is similar to the previous example, except that `s` contains a single binary real number (in IEEE format), rather an array of binary integers. The format string in each `Scan` call indicates that the source (`s`) is to be treated as a 1-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, the `z` modifier is not accepted.

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 the first 5 bytes of `s` to be ignored.

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);
```

Remarks

`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);
```

Remarks

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`, an error is reported. 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 the comma separators to be discarded.

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);
```

Remarks

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` = 200 (100 integers, 2 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);
```

Remarks

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);
}
```

Remarks

This example shows how a subroutine can be used 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. The two asterisks (`*`) in the format string are matched to `count`. For instance, if `count` is 100, then 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);
}
```

Remarks

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 the loop is exited. Otherwise, the prompt string is repeated.

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 be repeated at the beginning of each line until the user enters an integer value. Second, any characters entered after the integer value are discarded.

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);
}
```

Remarks

This example is similar to the previous example, except that the item being read from the keyboard is a string instead of an integer. The `w` modifier is used to prevent `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, see the *String to String* example earlier in this section.

Reading a Line from the Standard Input

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

Remarks

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. This can be done via the Formatting and I/O Library function 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 at least one less than the size of the buffer.

ReadLine transfers every character from the input line to the buffer, including leading, embedded, and trailing spaces, until the maximum number of bytes (for example, 80) have been transferred. Any remaining characters at the end of the line are discarded. The linefeed is never transferred to the buffer.

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

Chapter 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 one-dimensional (1D) and two-dimensional (2D) array manipulation, complex operations, matrix operations, and statistics. This section contains general information about the Analysis Library functions and panels.

The Analysis Library Function Panels

The Analysis Library function panels are grouped in a tree structure according to the types of operations performed. The Analysis Library function tree is shown in Table 3-1.

The first- and second-level bold 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 in plain text are the names of individual function panels. Each analysis function panel generates one analysis function call. The names of the corresponding analysis function calls appear in bold italics to the right of the function panel names.

Table 3-1. The Analysis Library Function Tree

Analysis	
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***

(continues)

Table 3-1. The Analysis Library Function Tree (Continued)

1D Linear Evaluation	***LinEv1D***
1D Maximum & Minimum	***MaxMin1D***
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***
Matrix Multiplication	***MatrixMul***
Matrix Inversion	***InvMatrix***
Transpose	***Transpose***
Determinant	***Determinant***
Array Utilities	
Clear Array	***Clear1D***
Set Array	***Set1D***
Copy Array	***Copy1D***
Get Error String	***GetAnalysisErrorString***

The classes and subclasses in the function tree are described here.

- 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 real and imaginary parts of complex numbers are processed 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 operating each function panel.

Hints for Using Analysis Function Panels

With the analysis function panels, you can manipulate scalars and arrays of data interactively. You will 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 things.

- The processing speed of the analysis functions is affected by the computer on which you are running LabWindows/CVI. 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 inordinate amount of time, keep the constraints of your hardware in mind.
- Many analysis routines for arrays run in place. That is, the input and output data can be stored in the same array. This is very important to keep in mind when you are processing

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.

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, **status** is set to the appropriate error value. Error messages corresponding to the possible **status** values are listed at the end of this chapter.

Analysis Library Function Reference

This section describes each function in the LabWindows/CVI Analysis Library. The LabWindows/CVI Analysis Library functions are arranged alphabetically.

Abs1D

`int` **status = Abs1D (**`double` **inputArray**`[]`**,** `int` **numberofElements,**
`double` **outputArray**`[]`**);**

Purpose

Finds the absolute value of the **inputArray**. The function performs the operation in place; **inputArray** and **outputArray** can be the same array.

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements.
Output	**outputArray**	double-precision array	Absolute value of input array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Add1D

`int` **status = Add1D (**`double` **arrayX**`[]`**,** `double` **arrayY**`[]`**,** `int` **numberofElements,**
`double` **outputArray**`[]`**);**

Purpose

Adds one-dimensional (1D) arrays. The function obtains the ith element of the output array by using the following formula:

$$z_i = x_i + y_i$$

The function performs the operation in place; that is, **outputArray** can be the same array as either **arrayX** or **arrayY**.

Parameters

Input	**arrayX**	double-precision array	Input array.
	arrayY	double-precision array	Input array.
	numberofElements	integer	Number of elements to be added.
Output	**outputArray**	double-precision array	Result array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Add2D

`int` **status = Add2D (**`void` ***arrayX,** `void` ***arrayY,** `int` **numberofRows,**
`int` **numberofColumns,** `void` ***outputArray);**

Purpose

Adds two (2D) arrays. The function obtains the (ith, jth) element of the output array by using the following formula.

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

The function performs the operation in place; **outputArray** can be the same array as either **arrayX** or **arrayY**.

Parameters

Input	**arrayX**	double-precision 2D array	Input array.
	arrayY	double-precision 2D array	Input array.
	numberofRows	integer	Number of elements in first dimension.
	numberofColumns	integer	Number of elements in second dimension.
Output	**outputArray**	double-precision 2D array	Result array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Clear1D

`int` **status = Clear1D (**`double` **array**`[]`**,** `int` **numberofElements);**

Purpose

Sets the elements of the **array** to zero.

Parameters

Input	**numberofElements**	integer	Number of elements in **array**.
Output	**array**	double-precision array	Cleared array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Copy1D

`int` **status = Copy1D (**`double` **inputArray**`[]`**,** `int` **numberofElements,**
`double` **outputArray**`[]`**);**

Purpose

Copies the elements of the **inputArray**. This function is useful to duplicate arrays for in-place operations.

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements in **inputArray**.
Output	**outputArray**	double-precision array	Duplicated array.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxAdd

`int` **status = CxAdd (**`double` **xReal,** `double` **xImaginary,** `double` **yReal,**
`double` **yImaginary,** `double` ***outputReal**
`double`***outputImaginary);**

Purpose

Adds two complex numbers. The function obtains the resulting complex number by using the formulas.

$zr = xr + yr$

$zi = xi + yi$

Parameters

Input	**xReal**	double-precision	Real part of x.
	xImaginary	double-precision	Imaginary part of x.
	yReal	double-precision	Real part of y.
	yImaginary	double-precision	Imaginary part of y.
Output	**outputReal**	double-precision	Real part of z.
	outputImaginary	double-precision	Imaginary part of z.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxAdd1D

`int` **status = CxAdd1D** (`double` **arrayXReal**`[]`, `double` **arrayXImaginary**`[]`,
`double` **arrayYReal**`[]`, `double` **arrayYImaginary**`[]`,
`int` **numberofElements**, `double` **outputArrayReal**`[]`,
`double` **outputArrayImaginary**`[]`);

Purpose

Adds two 1D complex arrays. The function obtains the ith element of the resulting complex array by using the following formulas.

$$zr_i = xr_i + yr_i$$

$$zi_i = xi_i + yi_i$$

The function performs the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**arrayXReal**	double-precision array	Real part of x.
	arrayXImaginary	double-precision array	Imaginary part of x.
	arrayYReal	double-precision array	Real part of y.
	arrayYImaginary	double-precision array	Imaginary part of y.
	numberofElements	integer	Number of elements.
Output	**outputArrayReal**	double-precision array	Real part of z.
	outputArrayImaginary	double-precision array	Imaginary part of z.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxDiv

`int` **status** = **CxDiv** (`double` **xReal**, `double` **xImaginary**, `double` **yReal**, **yImaginary**, `double` ***outputReal**, `double` ***outputImaginary**);

Purpose

Divides two complex numbers. The function obtains the resulting complex number by using the following formulas.

$$zr = (xr*yr + xi*yi) / (yr^2 + yi^2)$$

$$zi = (xi*yr - xr*yi) / (yr^2 + yi^2)$$

Parameters

Input	**xReal**	double-precision	Real part of x.
	xImaginary	double-precision	Imaginary part of x.
	yReal	double-precision	Real part of y.
	yImaginary	double-precision	Imaginary part of y.
Output	**outputReal**	double-precision	Real part of z.
	outputImaginary	double-precision	Imaginary part of z.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxDiv1D

```
int status = CxDiv1D (double arrayXReal[], double arrayXImaginary[],
                      double arrayYReal[], double arrayYImaginary)[],
                      int numberofElements, double outputArrayReal[],
                      double outputArrayImaginary[]);
```

Purpose

Divides two 1D complex arrays. The function obtains the ith element of the resulting complex array by using the following formulas.

$$zr_i = (xr_i * yr_i + xi_i * yi_i) / (yr_i^2 + yi_i^2)$$

$$zi_i = (xi_i * yr_i - xr_i * yi_i) / (yr_i^2 + yi_i^2)$$

The function performs the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**arrayXReal**	double-precision array	Real part of x.
	arrayXImaginary	double-precision array	Imaginary part of x.
	arrayYReal	double-precision array	Real part of y.
	arrayYImaginary	double-precision array	Imaginary part of y.
	numberofElements	integer	Number of elements.
Output	**outputArrayReal**	double-precision array	Real part of z.
	outputArrayImaginary	double-precision array	Imaginary part of z.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxLinEv1D

```
int status = CxLinEv1D (double arrayXReal[], double arrayXImaginary[],
                        int numberofElements, double aReal, double aImaginary,
                        double bReal, double bImaginary,
                        double outputArrayReal[],
                        double outputArrayImaginary[]);
```

Purpose

Performs a complex linear evaluation of a 1D complex array. The function obtains the ith element of the resulting complex array by using the following formulas.

$$yr_i = (ar * xr_i - ai * xi_i) + br$$

$$yi_i = (ar * xi_i + ai * xr_i) + bi$$

The function performs the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**arrayXReal**	double-precision array	Real part of x.
	arrayXImaginary	double-precision array	Imaginary part of x.
	numberofElements	integer	Number of elements.
	aReal	double-precision	Real part of a.
	aImaginary	double-precision	Imaginary part of a.
	bReal	double-precision	Real part of b.
	bImaginary	double-precision	Imaginary part of b.
Output	**outputArrayReal**	double-precision array	Real part of y.
	outputArrayImaginary	double-precision array	Imaginary part of y.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxMul

`int` **status** = **CxMul** (`double` **xReal**, `double` **xImaginary**, `double` **yReal**,
`double` **yImaginary**, `double` ***outputReal**,
`double` ***outputImaginary**);

Purpose

Multiplies two complex numbers. The function obtains the resulting complex number by using the following formulas.

$zr = xr*yr - xi*yi$

$zi = xr*yi + xi*yr$

Parameters

Input	**xReal**	double-precision	Real part of x.
	xImaginary	double-precision	Imaginary part of x.
	yReal	double-precision	Real part of y.
	yImaginary	double-precision	Imaginary part of y.
Output	**outputReal**	double-precision	Real part of z.
	outputImaginary	double-precision	Imaginary part of z.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxMul1D

`int` **status** = **CxMul1D** (`double` **arrayXReal**`[]`, `double` **arrayXImaginary**`[]`,
`double` **arrayYReal**`[]`, `double` **arrayYImaginary**`[]`,
`int` **numberofElements**, `double` **outputArrayReal**`[]`,
`double` **outputArrayImaginary**`[]`);

Purpose

Multiplies two 1D complex arrays. The function obtains the ith element of the resulting complex array by using the formulas:

$zr_i = xr_i * yr_i - xi_i * yi_i$

$zi_i = xr_i * yi_i + xi_i * yr_i$

The function performs the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**arrayXReal**	double-precision array	Real part of x.
	arrayXImaginary	double-precision array	Imaginary part of x.
	arrayYReal	double-precision array	Real part of y.
	arrayYImaginary	double-precision array	Imaginary part of y.
	numberofElements	integer	Number of elements.
Output	**outputArrayReal**	double-precision array	Real part of z.
	outputArrayImaginary	double-precision array	Imaginary part of z.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxRecip

`int` **status** = **CxRecip** (`double` **xReal**, `double` **xImaginary**, `double` ***outputReal**,
`double` ***outputImaginary**);

Purpose

Finds the reciprocal of a complex number. The function obtains the resulting complex number by using the following formulas.

$$yr = xr / (xr^2 + xi^2)$$

$$yi = -xi / (xr^2 + xi^2)$$

Parameters

Input	**xReal**	double-precision	Real part of x.
	xImaginary	double-precision	Imaginary part of x.
Output	**outputReal**	double-precision	Real part of y.
	outputImaginary	double-precision	Imaginary part of y.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxSub

`int` **status = CxSub** (`double` **xReal**, `double` **xImaginary**, `double` **yReal**,
`double` **yImaginary**, `double` ***outputReal**,
`double` ***outputImaginary**);

Purpose

Subtracts two complex numbers. The function obtains the resulting complex number by using the following formulas.

$$zr = xr - yr$$

$$zi = xi - yi$$

Parameters

Input	**xReal**	double-precision	Real part of x.
	xImaginary	double-precision	Imaginary part of x.
	yReal	double-precision	Real part of y.
	yImaginary	double-precision	Imaginary part of y.
Output	**outputReal**	double-precision	Real part of z.
	outputImaginary	double-precision	Imaginary part of z.

Return Value

status	integer	Refer to error codes in Table 3-2.

CxSub1D

```
int status = CxSub1D (double arrayXReal[], double arrayXImaginary[],
                      double arrayYReal[], double arrayYImaginary[],
                      int numberofElements, double outputArrayReal[],
                      double outputArrayImaginary[]);
```

Purpose

Subtracts two 1D complex arrays. The function obtains the ith element of the resulting complex array by using the following formulas.

$$zr_i = xr_i - yr_i$$

$$zi_i = xi_i - yi_i$$

The function performs the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**arrayXReal**	double-precision array	Real part of x.
	arrayXImaginary	double-precision array	Imaginary part of x.
	arrayYReal	double-precision array	Real part of y.
	arrayYImaginary	double-precision array	Imaginary part of y.
	numberofElements	integer	Number of elements.
Output	**outputArrayReal**	double-precision array	Real part of z.
	outputArrayImaginary	double-precision array	Imaginary part of z.

Return Value

status	integer	Refer to error codes in Table 3-2.

Determinant

`int` **status = Determinant** (`void` ***inputMatrix**, `int` **matrixSize**, `double` ***determinant**);

Purpose

Finds the determinant of a **matrixSize** by **matrixSize** 2D input matrix.

Parameters

Input	**inputMatrix**	double-precision 2D array	Input matrix.
	matrixSize	integer	Dimension size of input matrix.
Output	**determinant**	double-precision	Determinant.

Note: ***The input matrix must be a*** **matrixSize** ***by*** **matrixSize** ***square matrix.***

Return Value

status	integer	Refer to error codes in Table 3-2.

Div1D

`int` **status = Div1D** (`double` **arrayX**`[]`, `double` **arrayY**`[]`, `int` **numberofElements**, `double` **outputArray**`[]`);

Purpose

Divides two 1D arrays. The function obtains the ith element of the output array by using the following formula.

$$z_i = x_i / y_i$$

The function performs the operation in place; that is, **outputArray** can be the same array as either **arrayX** or **arrayY**.

Parameters

Input	**arrayX**	double-precision array	Input array.
	arrayY	double-precision array	Input array.
	numberofElements	integer	Number of elements to be divided.
Output	**outputArray**	double-precision array	Result array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Div2D

`int` **status = Div2D** (`void` ***arrayX**, `void` ***arrayY**, `int` **numberofRows**,
`int` **numberofColumns**, `void` ***outputArray**);

Purpose

Divides two 2D arrays. The function obtains the (ith, jth) element of the output array by using the following formula.

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

The function performs the operation in place; that is, **outputArray** can be the same array as either **arrayX** or **arrayY**.

Parameters

Input	**arrayX**	double-precision 2D array	Input array.
	arrayY	double-precision 2D array	Input array.
	numberofRows	integer	Number of elements in first dimension.
	numberofColumns	integer	Number of elements in second dimension.
Output	**outputArray**	double-precision 2D array	Result array.

Return Value

status	integer	Refer to error codes in Table 3-2.

DotProduct

`int` **status** = **DotProduct** (`double` **vectorX**`[]`, `double` **vectorY**`[]`,
`int` **numberofElements**,
`double` ***dotProduct**);

Purpose

Computes the dot product of the **vectorX** and **vectorY** input arrays. The function obtains the dot product by using the following formula:

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

Parameters

Input	**vectorX**	double-precision array	Input vector.
	vectorY	double-precision array	Input vector.
	numberofElements	integer	Number of elements.
Output	**dotProduct**	double-precision	Dot product.

Return Value

status	integer	Refer to error codes in Table 3-2.

GetAnalysisErrorString

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

Purpose

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

Parameters

Input	**errorNum**	integer	Status returned by an Analysis function.

Return Value

message	string	Explanation of error.

Histogram

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

Purpose

Computes the histogram of the **inputArray**. The histogram is obtained by counting the number of times that the elements in the input array fall in the ith interval. Let

$$\Delta x = (xTop - xBase) / intervals$$

$$y_{x,i} = \begin{cases} 1 & \text{if } i\Delta x \le x - xBase < (i + 1)\Delta x \\ 0 & \text{otherwise} \end{cases}$$

The ith element of the histogram is:

$$hist_i = \sum_{j=0}^{n-1} y(x_j, i)$$

The values of the histogram axis are the mid-point values of the intervals:

$$axis_i = i\Delta x + \Delta x / 2 + xBase$$

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements in **Input Array**.
	base	double-precision	Lower range.
	top	double-precision	Upper range.
	intervals	integer	Number of intervals.
Output	**histogramArray**	integer array	Histogram of **input Array**.
	axisArray	double-precision array	Histogram axis array.

Return Value

status	integer	Refer to error codes in Table 3-2.

InvMatrix

`int` **status** = **InvMatrix** (`void` ***inputMatrix**, `int` **matrixSize**, `void` ***outputMatrix**);

Purpose

Finds the inverse matrix of an input matrix. The operation can be performed in place; that is, **inputMatrix** and **outputMatrix** can be the same matrices.

Parameters

Input	**inputMatrix**	double-precision 2D array	Input matrix.
	matrixSize	integer	Dimension of matrix.
Output	**outputMatrix**	double-precision 2D array	Inverse matrix.

Note: ***The input matrix must be a*** **matrixSize** ***by*** **matrixSize** ***square matrix***.

Return Value

status	integer	Refer to error codes in n Table 3-2.

LinEv1D

`int` **status** = **LinEv1D** (`double` **inputArray**`[]`, `int` **numberofElements**,
`double` **multiplier**, `double` **additiveConstant**,
`double` **outputArray**`[]`);

Purpose

Performs a linear evaluation of a 1D array. The function obtains the ith element of the output array by using the following formula.

$$y_i = a * x_i + b$$

The operation can be performed in place; that is, **inputArray** and **outputArray** can be the same array.

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements.
	multiplier	double-precision	Multiplicative constant.
	additiveConstant	double-precision	Additive constant.
Output	**outputArray**	double-precision array	Linearly evaluated array.

Return Value

status	integer	Refer to error codes in Table 3-2.

LinEv2D

`int` **status** = **LinEv2D** (`void` ***inputArray**, `int` **numberofRows**, `int` **numberofColumns**,
`double` **multiplier**, `double` **additiveConstant**,
`void` ***outputArray**);

Purpose

Performs a linear evaluation of a 2D array. The function obtains the (ith, jth) element of the output array by using the following formula.

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

The function performs the operation in place; that is, **inputArray** and **outputArray** can be the same array.

Parameters

Input	**inputArray**	double-precision 2D array	Input array.
	numberofRows	integer	Number of elements in first dimension.
	numberofColumns	integer	Number of elements in second dimension.
	multiplier	double-precision	Multiplicative constant.
	additiveConstant	double-precision	Additive constant.
Output	**outputArray**	double-precision 2D array	Linearly evaluated array.

Return Value

status	integer	Refer to error codes in Table 3-2.

MatrixMul

`int` **status** = **MatrixMul** (`void` ***matrixX**, `void` ***matrixY**, `int` **#ofRowsInX**,
`int` **cols/rowsInX/Y**, `int` **#ofColumnsInY**,
`void` ***outputMatrix**);

Purpose

Multiplies two 2D input matrices. The function obtains the (ith, jth) element of the output matrix by using the following formula.

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

Parameters

Input	**matrixX**	double-precision 2D array	**matrixX** input matrix.
	matrixY	double-precision 2D array	**matrixY** input matrix.
	#ofRowsInX	integer	First dimension of **matrixX**.
	cols/rowsInX/Y	integer	Second dimension of **matrixX**.; first dimension of **matrixY**.
	#ofColumnsInY	integer	Second dimension of **matrixY**.
Output	**outputMatrix**	double-precision 2D array	Output matrix.

Return Value

status	integer	Refer to error codes in Table 3-2.

Parameter Discussion

Note: ***Be sure to use the correct array sizes. The following array sizes must be met:***

- **matrixX** ***must be*** (**#ofRowsInX** ***by*** **cols/rowsInX/Y**).
- **matrixY** ***must be*** (**cols/rowsInX/Y** ***by*** **#ofColumnsInY**).
- **outputMatrix** ***must be*** (**#ofRowsInX** ***by*** **#ofColumnsInY**).

MaxMin1D

```
int status = MaxMin1D (double inputArray[], int numberofElements,
                       double *maximumValue, int *maximumIndex,
                       double *minimumValue, int *minimumIndex);
```

Purpose

Finds the maximum and minimum values in the input array, as well as the respective indices of the first occurrence of the maximum and minimum values.

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements.
Output	**maximumValue**	double-precision	Maximum value.
	maximumIndex	integer	Index of **maximumValue** in **inputArray**.
	minimumValue	double-precision	Minimum value.
	minimumIndex	integer	Index of **minimumValue** in **inputArray**.

Return Value

status	integer	Refer to error codes in Table 3-2.

MaxMin2D

```
int status = MaxMin2D (void *inputArray, int numberofRows,
                       int numberofColumns, double *maximumValue,
                       int *maximumRowIndex, int *maximumColumnIndex,
                       double *minimumValue, int *minimumRowIndex,
                       int *minimumColumnIndex);
```

Purpose

Finds the maximum and the minimum values in the 2D input array, as well as the respective indices of the first occurrence of the maximum and minimum values. The **inputArray** is scanned by rows.

Parameters

Input	**inputArray**	double-precision 2D array	Input array.
	numberofRows	integer	Number of elements in first dimension of **inputArray**.
	numberofColumns	integer	Number of elements in second dimension of **inputArray**.
Output	**maximumValue**	double-precision	Maximum value.
	maximumRowIndex	integer	Index of **maximumValue** in **inputArray** array (first dimension).
	maximumColumnIndex	integer	Index of **maximumValue** in **inputArray** (second dimension).
	minimumValue	double-precision	Minimum value.
	minimumRowIndex	integer	Index of **minimumValue** in **inputArray** (first dimension).
	minimumColumnIndex	integer	Index of **minimumValue** in **inputArray** array (second dimension).

Return Value

status	integer	Refer to error codes in Table 3-2.

Mean

`int` **status** = **Mean** (`double` **inputArray**`[]`, `int` **numberofElements**, `double` ***mean**);

Purpose

Compute the mean (average) value of the input array. The function uses the following formula to find the mean.

$$meanval = \sum_{i=0}^{n-1} x_i \; / \; n$$

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements in **inputArray**.
Output	**mean**	double-precision	Mean value.

Return Value

status	integer	Refer to error codes in Table 3-2.

Mul1D

`int` **status** = **Mul1D** (`double` **arrayX**`[]`, `double` **arrayY**`[]`, `int` **numberofElements**, `double` **outputArray**`[]`);

Purpose

Multiplies two 1D arrays. The function obtains the ith element of the output array by using the following formula.

$$z_i = x_i * y_i$$

The function performs the operation in place; that is, **outputArray** can be the same array as either **arrayX** or **arrayY**.

Parameters

Input	**arrayX**	double-precision array	Input array.
	arrayY	double-precision array	Input array.
	numberofElements	integer	Number of elements to be multiplied.
Output	**outputArray**	double-precision array	Result array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Mul2D

`int` **status** = **Mul2D** (`void` ***arrayX**, `void` ***arrayY**, `int` **numberofRows**, `int` **numberofColumns**, `void` ***outputArray**);

Purpose

Multiplies two 2D arrays. The function obtains the (ith, jth) element of the output array by using the following formula.

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

The function performs the operation in place; that is, **outputArray** can be the same array as either **arrayX** or **arrayY**.

Parameters

Input	**arrayX**	double-precision 2D array	Input array.
	arrayY	double-precision 2D array	Input array.
	numberofRows	integer	Number of elements in first dimension.
	numberofColumns	integer	Number of elements in second dimension.
Output	**outputArray**	double-precision 2D array	Result array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Neg1D

`int` **status** = **Neg1D** (`double` **inputArray**`[]`, `int` **numberofElements**,
`double` **outputArray**`[]`);

Purpose

Negates the elements of the input array. The function performs the operation in place; that is, **inputArray** and **outputArray** can be the same array.

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements.
Output	**outputArray**	double-precision array	Negated values of the **inputArray**.

Return Value

status	integer	Refer to error codes in Table 3-2.

Set1D

`int` **status** = **Set1D** (`double` **array**`[]`, `int` **numberofElements**, `double` **setValue**);

Purpose

Sets the elements of the input array to a constant value.

Parameters

Input	**numberofElements**	integer	Number of elements in **array**.
	setValue	double-precision	Constant value.
Output	**array**	double-precision array	Result array (set to the value of **setValue**).

Return Value

status	integer	Refer to error codes in Table 3-2.

Sort

`int` **status** = **Sort** (`double` **inputArray**`[]`, `int` **numberofElements**, `int` **direction**,
`double` **outputArray**`[]`);

Purpose

Sorts the input array in ascending or descending order. The function performs the operation in place; that is, **inputArray** and **outputArray** can be the same array.

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements to be sorted.
	direction	integer	0: ascending. Non-zero: descending.
Output	**outputArray**	double-precision array	Sorted array.

Return Value

status	integer	Refer to error codes in Table 3-2.

StdDev

`int` **status** = **StdDev** (`double` **inputArray**`[]`, `int` **numberofElements**, `double` ***mean**,
`double` ***standardDeviation**);

Purpose

Computes the standard deviation and the mean (average) values of the input array. The formulas used to find the mean and the standard deviation are as follows.

$$meanval = \sum_{i=0}^{n-1} x_i \; / \; n$$

$$sDev = \sqrt{\sum_{i=0}^{n-1} [x_i - ave]^2 \; / \; n}$$

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements in **inputArray**.
Output	**mean**	double-precision	Mean value.
	standardDeviation	double-precision	Standard deviation.

Return Value

status	integer	Refer to error codes in Table 3-2.

Sub1D

`int` **status = Sub1D** (`double` **arrayX**`[]`, `double` **arrayY**`[]`, `int` **numberofElements**, `double` **outputArray**`[]`);

Purpose

Subtracts two 1D arrays. The function obtains the ith element of the output array by using the following formula:

$$z_i = x_i - y_i$$

The operation can be performed in place; that is, **outputArray** can be in place of either **arrayX** or **arrayY**.

Parameters

Input	**arrayX**	double-precision array	Input array.
	arrayY	double-precision array	Input array.
	numberofElements	integer	Number of elements to be subtracted.
Output	**outputArray**	double-precision array	Result array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Sub2D

`int` **status = Sub2D** (`void` ***arrayX**, `void` ***arrayY**, `int` **numberofRows**, `int` **numberofColumns**, `void` ***outputArray**);

Purpose

Subtracts two 2D arrays. The function obtains the (ith, jth) element of the output array by using the formula:

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

The function performs the operation in place; that is, **outputArray** can be in place of either **arrayX** or **arrayY**.

Parameters

Input	**arrayX**	double-precision 2D array	Input array.
	arrayY	double-precision 2D array	Input array.
	numberofRows	integer	Number of elements in first dimension.
	numberofColumns	integer	Number of elements in second dimension.
Output	**outputArray**	double-precision 2D array	Result array.

Return Value

status	integer	Refer to error codes in Table 3-2.

Subset1D

`int` **status = Subset1D** (`double` **inputArray**`[]`, `int` **numberofElements**, `int` **index**, `int` **length**, `double` **outputArray**`[]`);

Purpose

Extracts a subset of the **inputArray** input array containing the number of elements specified by the **length** and starting at the **index** element.

Parameters

Input	**inputArray**	double-precision array	Input array.
	numberofElements	integer	Number of elements in **inputArray**.
	index	integer	Initial index for the subset.
	length	integer	Number of elements copied to the subset.
Output	**outputArray**	double-precision array	Subset array.

Return Value

status	integer	Refer to error codes in Table 3-2.

ToPolar

`int` **status = ToPolar** (`double` **xReal**, `double` **yImaginary**, `double` ***magnitude**, `double` ***phaseRadians**);

Purpose

Converts the rectangular coordinates (**xReal**, **yImaginary**) to polar coordinates (**magnitude**, **phaseRadians**). The formulas used to obtain the polar coordinates are as follows.

$$mag = \sqrt{x^2 + y^2}$$

$$phase = arctan\ (y/x)$$

The **phaseRadians** value is in the range of [$-\pi$ to π]

Parameters

Input	**xReal**	double-precision	X coordinate.
	yImaginary	double-precision	X coordinate.
Output	**magnitude**	double-precision	Magnitude.
	phaseRadians	double-precision	Phase (in radians).

Return Value

status	integer	Refer to error codes in Table 3-2.

ToPolar1D

```
int status = ToPolar1D (double arrayXReal[], double arrayYImaginary[],
                        int numberofElements, double magnitude[],
                        double phaseRadians[]);
```

Purpose

Converts the set of rectangular coordinate points (**arrayXReal**, **arrayYImaginary**) to a set of polar coordinate points (**magnitude**, **phaseRadians**). The function obtains the ith element of the polar coordinate set by using the following formulas.

$$mag_i = \sqrt{x_i^2 + y_i^2}$$

$$phase_i = arctan\ y_i / x_i$$

The **phaseRadians** value is in the range of [$-\pi$ to π].

The function performs the operations in place; that is, **arrayXReal** and **magnitude**, and **arrayYImaginary** and **phaseRadians**, can be the same arrays, respectively.

Parameters

Input	**arrayXReal**	double-precision array	X coordinate.
	arrayYImaginary	double-precision array	Y coordinate.
	numberofElements	integer	Number of elements.
Output	**magnitude**	double-precision array	Magnitude.
	phaseRadians	double-precision array	Phase (in radians).

Return Value

status	integer	Refer to error codes in Table 3-2.

ToRect

`int` **status = ToRect** (`double` **magnitude**, `double` **phaseRadians**, `double` ***xReal**,
`double` ***yImaginary**);

Purpose

Converts the polar coordinates (**magnitude**, **phaseRadians**) to rectangular coordinates (**xReal**, **yImaginary**). The formulas used to obtain the rectangular coordinates are as follows.

$x = mag * cos(phase)$

$y = mag * sin(phase)$

Parameters

Input	**magnitude**	double-precision	Magnitude.
	phaseRadians	double-precision	Phase (in radians).
Output	**xReal**	double-precision	X coordinate.
	yImaginary	double-precision	Y coordinate.

Return Value

status	integer	Refer to error codes in Table 3-2.

ToRect1D

```
int status = ToRect1D (double magnitude[], double phaseRadians[],
                       int numberofElements, double outputArrayReal[],
                       double outputArrayImaginary[]);
```

Purpose

Converts the set of polar coordinate points (**magnitude**, **phaseRadians**) to a set of rectangular coordinate points (**outputArrayReal**, **outputArrayImaginary**). The function obtains the ith element of the rectangular set by using the following formulas.

$$x_i = mag_i * \cos(phase_i)$$

$$y_i = mag_i * \sin(phase_i)$$

The function performs the operations in place; that is, **outputArrayReal** and **magnitude**, and **outputArrayImaginary** and **phaseRadians**, can be the same arrays, respectively.

Parameters

Input	**magnitude**	double-precision array	Magnitude.
	phaseRadians	double-precision array	Phase (in radians).
	numberofElements	integer	Number of elements.
Output	**outputArrayReal**	double-precision array	X coordinate.
	outputArrayImaginary	double-precision array	Y coordinate.

Return Value

status	integer	Refer to error codes in Table 3-2.

Transpose

`int` **status = Transpose** (`void` ***inputMatrix**, `int` **numberofRows**,
`int` **numberofColumns**, `void` ***outputMatrix**);

Purpose

Finds the transpose of a 2D input matrix. The (ith, jth) element of the resulting matrix uses the formula:

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

Parameters

Input	**inputMatrix**	double-precision 2D array	Input matrix.
	numberofRows	integer	Size of first dimension.
	numberofColumns	integer	Size of second dimension.
Output	**outputMatrix**	double-precision 2D array	Transpose matrix.

Note: ***If the input matrix is dimensioned (*numberofRows *by* numberofColumns*), then the output matrix must be dimensioned (*numberofColumns *by* numberofRows*).***

Return Value

status	integer	Refer to error codes in Table 3-2.

Error Conditions

If an error condition occurs during a call to any of the functions in the LabWindows/CVI Analysis Library, the status return value contains the error code. This code is a value that specifies the type of error that occurred. The currently defined error codes and their associated meanings are given in Table 3-2.

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 err.
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.

Chapter 4
GPIB/GPIB-488.2 Library

This describes the NI-488 and NI-488.2 functions in the LabWindows/CVI GPIB Library, as well as the Device Manager functions in LabWindows/CVI. 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. Detailed descriptions of the NI-488 and NI-488.2 functions can be found in your NI-488.2 function reference manual. 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.

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 board.

GPIB Functions Library Function Panels

The GPIB Functions Library function panels are grouped in a tree structure according to the types of operations performed. The GPIB Functions Library function tree is in Table 4-1.

The first- and second-level bold headings in the function tree are names of the function classes. Function classes are groups of related function panels. The third-level headings in plain text are the names of individual function panels. Each GPIB function panel generates a GPIB function call. The actual function names are in bold italics in columns to the right.

Table 4-1. The GPIB Functions Library Function Tree

GPIB/GPIB-488.2 Library	
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***
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***

(continues)

Table 4-1. The GPIB Functions Library Function Tree (Continued)

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	***ibsignal***
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***
Callbacks (Windows only)	
Install Synchronous Callback	***ibInstallCallback***
Install Asynchronous Callback	***ibNotify***
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 Devices	***SendList***
Receive	***Receive***
Trigger and Clear	
Trigger Device	***Trigger***
Trigger Multile Devices	***TriggerList***
Clear Device	***DevClear***
Clear Multiple Devices	***DevClearList***

(continues)

Table 4-1. The GPIB Functions Library Function Tree (Continued)

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	***FindLstn***
Pass Control	***PassControl***
Low-Level I/O	
Send Commands	***SendCmds***
Setup for Sending	***SendSetup***
Send Data Bytes	***SendDataBytes***
Setup for Receiving	***ReceiveSetup***
Receive Response Message	***RcvRespMsg***

The classes and subclasses in the tree are described here.

- The **Open/Close** function panels open and close GPIB boards and devices.
- The **Configuration** function panels alter configuration parameters that were 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. These functions can be used at either 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 board. These functions are normally used when the GPIB board is not controller-in-charge.
- The **Callbacks** function panels install callback functions that are invoked when certain GPIB events occur. The functions in this class are available only under Windows. Under UNIX, you can use the `ibsgnl` function.
- 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 needed only for multithreaded applications and are available only on Windows 95 and 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 via the GPIB or locally via 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.

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 that the GPIB Library contains.

GPIB Libraries and the GPIB Dynamic Link Library/Device Driver

LabWindows/CVI for Windows uses National Instruments standard Windows `GPIB.DLL`. LabWindows/CVI for Sun uses the standard Sun Solaris-installed GPIB device drivers. These

drivers are packaged with your GPIB interface board 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 board documentation.

You can use a utility program 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 are using the LabWindows/CVI GPIB Library function panels, parameters that you specified using `IBCONF` are still in effect. You can also modify configuration parameters directly from one of the LabWindows/CVI configuration function panels, or from your program.

If you are using a LabWindows/CVI Instrument Library module, you do not need to make any changes using `IBCONF`. The module takes into account any special configuration requirements for the instrument controlled by the module. 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 either the `OpenDev`, the `ibfind`, or the `ibdev` function. Instrument modules must use the `OpenDev` function. When you open a device, an integer value representing a device descriptor is returned. All subsequent operations that involve a particular device require that you specify this device descriptor.
- If `OpenDev` is used, the `CloseDev` function should be used 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 must also 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 by activities such as reading from and writing to devices or polling them for status. Device functions access a specific device and take care of the addressing and bus management protocol for that device. Because they execute automatically, 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 may 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 board directly and require you to do 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 status bytes of a device you can enable. 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. Queues are necessary because some devices can send multiple positive status bytes back-to-back. When a positive response from a device is received, the RQS bit of its status word (`ibsta`) is set. The polling continues until SRQ is unasserted or an error condition is detected.

If the handler cannot locate the device requesting service (no known device responds positively to the poll) or if SRQ becomes stuck on (because of a faulty instrument or cable), a GPIB system error exists that will interfere with the proper evaluation of the RQS bit in the status words of devices. The error ESRQ is reported to you when you issue an `ibwait` call with the RQS bit

included in the wait mask. Aside from the difficulty caused by ESRQ in waiting for RQS, the error will have no detrimental effects on other GPIB operations.

If you call the serial poll function `ibrsp` and have received one or more responses previously via 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. A 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 have been 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 is still set after you call*** `ibrsp`**, t*he response byte queue has at least one more response byte remaining. You should call*** `ibrsp` ***until RQS is cleared to gather all stored response bytes and to guard against queue overflow*.**

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 both board calls and device calls. If you choose to look for SRQI in your program, you must disable Autopolling.

Board functions are also 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 have disabled the interrupts of the GPIB interface board via `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 has completed and is about to return to the application program.

If you have enabled hardware interrupts, the handler can respond to SRQI interrupts and perform Autopolling even when the handler is not performing a function. However, the handler will not conduct an Autopoll if any of the following conditions exist.

- The last GPIB call was a board call. Autopolling is re-instated after a subsequent device call.
- GPIB I/O is in progress. In particular, during asynchronous GPIB I/O, autopolling will not occur until the asynchronous I/O has completed.
- The "stuck SRQ" condition exists.
- Autopolling has been disabled by `IBCONF` or by `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. By prior arrangement, the Listener stops 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, the Listener must be properly configured to unambiguously detect the end of a transmission.

Using the configuration program, you can accommodate all permissible forms of read and write termination. (You cannot force the handler to ignore END on read operations.) The default configuration settings for read and write termination can also be changed at run time using the `ibeos` and `ibeot` functions.

Timeouts

A timeout mechanism regulates the GPIB routines that transfer command sequences or data messages. A default timeout period of 10 sec is preconfigured in the handler; thus, all I/O must complete within that period to avoid a timeout error. The default timeout value can be changed with the IBCONF utility. In addition, you can use the NI-488 board function call `ibtmo` to programmatically alter the timeout period.

Regardless of the I/O and Wait timeout period, a much shorter timeout is enforced for responses to serial polls. This shorter timeout period takes effect whenever a serial poll is conducted. Because devices normally respond quickly to polls, there is no need to wait the relatively lengthy I/O timeout period for a nonresponsive device.

Global Variables for the GPIB Library

The following global variables are used by the GPIB Library and the GPIB-488.2 Library:

- Status Word (`ibsta`)
- Error (`ibcnt`, `ibcntl`)

These variables are updated 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.

Different Levels of Functionality Depending on Platform and GPIB Board

In general, the GPIB library is same for all platforms and GPIB boards. There are, however, some exceptions, most notably relating to SRQ notification, support for multithreading, and limitations on transfer size. These particular issues are discussed later in this chapter. This section explains the various categories of GPIB support.

Windows 95

There are two kinds of GPIB support for Windows 95. The “native 32-bit” driver and the “compatibility” driver. You can see which one you have installed on your system by running the GPIB Information program in your GPIB Software group and noting the name of the driver.

Driver Name	Description
NI-488.2M	Native 32-bit driver.
NI-488.2	Compatibility driver.

Native 32-Bit Driver

The native 32-bit driver is a 32-bit device driver written specifically for Windows 95. It is supported on the following boards.

- AT-GPIB/TNT
- AT-GPIB/TNT+
- AT-GPIB/TNT (PnP)
- PCI-GPIB
- PCMCIA-GPIB
- PCMCIA-GPIB+

If you want to use GPIB under Windows 95 and you have an older board, it is recommended that you upgrade to one of the boards in this list.

Compatibility Driver

The compatibility driver is a 32-to-16-bit thunking DLL that you can use with the Windows 3.1 GPIB driver under Windows 95. All GPIB boards are supported by the compatibility driver. The compatibility driver has several limitations. In particular, it does not support multithreading and transfers are limited to 64k bytes.

Windows NT

The GPIB driver for Windows NT is a native 32-bit driver written specifically for Windows NT. Version 1.0 supports the following boards:

- AT-GPIB
- AT-GPIB/TNT

Version 1.2, due to be released in the second half of 1996, will add support for the PCI-GPIB and PCMCIA-GPIB.

Limitations on Transfer Size

There are no limitations on transfer size except for the compatibility driver under Windows 95. The compatibility driver is limited to 64 KB transfers.

Multithreading

If you are using multithreading in an external compiler, you can call GPIB functions from more than one thread at the same time under Windows NT or under Windows 95 with the native 32-bit driver. In order to be truly multithreaded safe, you must use on of the following versions of the GPIB 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 the `ThreadIbsta`, `ThreadIberr`, `ThreadIbcnt`, or `ThreadIbcntl` functions. 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 they are maintained on a *per process* basis.

Notification of SRQ and Other GPIB Events

Synchronous Callbacks

Under Windows 3.1, you can use `ibInstallCallback` to specify a function to be called when an SRQ is asserted on the GPIB or when an asynchronous I/O operation has completed. It is a board-level function only.

The same functionality exists on Windows 95 when you are using the compatibility driver.

If you are using Windows NT or the native 32-bit driver for Windows 95, you can use `ibInstallCallback` to specify functions to be invoked on the occurrence of any board-level or device-level condition on which you can wait using the `ibwait` function.

Callback functions installed with `ibInstallCallback` are *synchronous* callbacks, that is, they are invoked only when LabWindows/CVI is processing 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

You have the ability to install *asynchronous* callbacks on Windows NT and on Windows 95 with the native 32-bit driver. Asynchronous callbacks are installed with the `ibnotify` function and can be called 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. See the documentation of the `ibnotify` function later in this chapter for more details.

Driver Version Requirements

If you are using Windows NT, you must have version 1.2 or later of the GPIB driver to use the `ibInstallCallback` and `ibnotify` functions.

If you are using the native 32-bit GPIB driver on Windows 95, you must have version 1.1 or later to use the `ibInstallCallback` and `ibnotify` functions.

If you are using the Windows 3.1 compatibility driver on Windows 95, you can use the limited version of `ibInstallCallback`, but you cannot use `ibnotify`.

GPIB Function Reference

Most of the functions in the GPIB/GPIB-488.2 Library are described in the software reference manual that you received with your GPIB board. This section contains descriptions 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	**Device**	integer	The device to be closed.

Return Value

result	integer	Result of the close device operation.

Return Codes

`-1`	Error—cannot find device.
`0`	Success.

Using This Function

Takes a device offline. `CloseDev` performs an `ibloc`, then an `ibonl` with a value of zero. **Device** is the device descriptor returned when the device was opened with `OpenDev`. If `CloseDev` cannot find the device descriptor in its table, a `-1` is returned. `CloseDev` should be used only in conjunction with `OpenDev`. Never call `CloseDev` with a device descriptor obtained by calling `ibfind`.

CloseInstrDevs

`int` **result = CloseInstrDevs** (`char` ***instrumentPrefix**);

Purpose

Closes instrument devices.

Parameter

Input	**instrumentPrefix**	string	Must be null-terminated.

Return Value

result	integer	Result of the close instrument devices operation.

Return Codes

`0`	Success.

Using This Function

Closes all devices associated with the instrument module whose prefix is specified. **instrumentPrefix** is a string that specifies the prefix of the instrument module being closed. `CloseInstrDevs` always returns zero. `CloseInstrDevs` should be used only in conjunction with `OpenDev`.

---

ibInstallCallback

`int` **status = ibInstallCallback** (`int` **boardOrDevice**, `int` **eventMask**,
`GPIBCallbackPtr` **callbackFunction**,
`void *`**callbackData)**

Note: ***This function is available only on Microsoft Windows. On UNIX, use the*** `ibsgnl` ***function. On Windows 3.1, the data type of the return value and the first two parameters is*** `short` ***rather than*** `int`**.**

Purpose

This function allows you to install a synchronous callback function for a specified board or device. If you want to install an asynchronous callback, use the `ibnotify` function instead.

The callback function is called when any of the GPIB events specified in the Event Mask parameter have occurred on the board 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 have called `RunUserInterface` and none of your callback functions are currently active. The callbacks are termed "synchronous" because they can be invoked only in the context of normal event processing.

Unlike asynchronous callbacks, there are no restrictions 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.

Only one callback function can apply for each board or device. Each call to this function for the same board or device supersedes the previous call.

To disable callbacks for a board or device, pass 0 for the **event Mask** parameter.

To use this function with the NI-488.2M (native 32-bit) driver, you must have one of the following versions.

- For Windows 95: Version 1.1 or later.
- For Windows NT: Version 1.2 or later.

If you use the NI-488.2 driver (the Windows 3.1 driver or the compatibility driver in Windows 95), you must pass a board index for the first parameter, and you can use only `SRQI` or `CMPL` for the event mask parameter.

Parameters

Input	**boardOrDevice**	integer (short integer on Windows 3.1)	A board index, or a board or device descriptor returned by `OpenDev`, `ibfind`, or `ibdev`. (On 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. See discussion below.
	callbackFunction	GPIBCallbackPtr	The name of the user function that is called when the specified events occur. See discussion below.
	callbackData	void pointer	A pointer to a user-defined four-byte value that is passed to the callback function.

Return Value

status	integer (short integer on Windows 3.1)	The 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` status variable.

eventMask

The conditions upon which to invoke the callback function are specified as bits in the **eventMask** parameter. The bits corresponds 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 have set in the mask is already TRUE, the callback is scheduled immediately. For example, if you pass `CMPL` as the **eventMask**, and the `ibwait` function would currently return a status word with `CMPL` set, the callback is scheduled immediately.

If you are using a NI-488.2M (native 32-bit) driver then the following mask bits are valid:

- At the board level, you can specify any of the status word bits that can be specified in the **waitMask** parameter to the `ibwait` function 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 can be specified in the **waitMask** parameter to the `ibwait` function for a device, other than `ERR`. This includes `RQS`, `END`, `CMPL`, and `TIMO`.

If you are using a NI-488.2 driver (Windows 3.1 or compatibility driver for Windows 95), then the only following mask bits are valid:

`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, Auto Serial Polling must be disabled. You can disable Auto Serial Polling with the following function call:

```
ibconfig (boardIndex, IbcAUTOPOLL, 0);
```

If you want to install a callback for the `RQS` (device-level) event, Auto Serial Polling must be enabled for the board. You can enable Auto Serial Polling 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**);

The **mask** and **callbackData** parameters are the same values that were passed to `ibInstallCallback`.

If invoked because of an `SRQI` or `RQS` condition, the callback function should call the `ibrsp` function 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 an asynchronous I/O operation (started by `ibrda`, `ibwrta`, or `ibcmda`) completed, the callback function should contain the following call:

```
ibwait (boardOrDevice, TIMO | CMPL);
```

The `ibcnt` and `ibcntl` status variables are not updated until this call to `ibwait` is made.

See Also

`ibnotify`

---

ibNotify

`int` **status = ibnotify** (`int` **boardOrDevice**, `int` **eventMask**,
`GpibNotifyCallback_t` **callbackFunction**, `void *`**callbackData**);

Note: ***This function is available only on Windows 95 and NT. On UNIX, use the*** `ibsgnl` ***function.***

Purpose

This function allows you to install an asynchronous callback function for a specified board or device. If you want to install a synchronous callback, use the `ibInstallCallback` function instead.

The callback function is called when any of the GPIB events specified in the **eventMask** parameter have occurred on the specified board or device. Asynchronous callbacks can be called 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. See the **Restrictions on Operations in Asynchronous Callbacks** discussion below.

Only one callback function can apply for each board or device. Each call to this function for the same board or device supersedes the previous call.

To disable callbacks for a board or device, pass 0 for the **eventMask** parameter.

Parameters

Input	**boardOrDevice**	integer	A board index, or a board or device descriptor returned by `OpenDev`, `ibfind`, or `ibdev`.
	eventMask	integer	Specifies the events upon which the callback function is called. Pass 0 to disable callbacks. See discussion below.
	callbackFunction	GpibNotifyCallback_t	The name of the user function that is called when the specified events occur. See discussion below.
	callbackData	void pointer	A pointer to a user-defined four-byte value that is passed to the callback function.

Return Value

status	integer	The same value as the `ibsta` status variable. Refer to your NI-488.2M user manual for a description of the values of `ibsta` status variable.

eventMask

The conditions upon which to invoke the callback function are specified as bits in the **eventMask** parameter. The bits corresponds 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 have set in the mask is already TRUE, the callback is called immediately. For example, if you pass `CMPL` as the **eventMask**, and the `ibwait` function would currently return a status word with `CMPL` set, the callback is called immediately.

At the board level, you can specify any of the status word bits that can be specified in the **waitMask** parameter to the `ibwait` function 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 can be specified in the **waitMask** parameter to the `ibwait` function 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, Auto Serial Polling must be disabled. You can disable Auto Serial Polling with the following function call:

```
ibconfig (boardIndex, IbcAUTOPOLL, 0);
```

If you want to install a callback for the RQS (device-level) event, Auto Serial Polling must be enabled for the board. You can enable Auto Serial Polling 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**);

The **callbackData** parameter is the same **callbackData** value passed to `ibInstallCallback`. The **sta**, **err**, and **cntl** parameters contain the information that you normally obtain using the `ibsta`, `iberr`, and `ibcntl` global variables or the `ThreadIbsta`, `ThreadIberr`, and `ThreadIbcntl` functions. The global variables and thread status functions return undefined values within the callback function. So you must use the **sta**, **err** and **cntl** parameters instead.

The value that you return from the callback function is very important. It is the event mask that is used to *rearm* the callback. If you return 0, the callback is disarmed (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`, the new event mask is used. 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`.

Warning: ***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 the `ibrsp` function to read the status byte. For an `SRQI` (board-level) condition, calling the `ibrsp` function is necessary to cause to requesting device to turn off the SRQ line.

```
char statusByte;
ibrsp (device, &statusByte);
```

If invoked because an asynchronous I/O operation (started by ibrda, ibwrta, or ibcmda) completed, the callback function should contain the following call:

```
ibwait (boardOrDevice, TIMO | CMPL);
```

The ibcnt and ibcntl status variables are not updated until this call to ibwait is made.

Restrictions on Operations in Asynchronous Callbacks

Callbacks installed with ibnotify can be called 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. You can do the following:

- Call the User Interface Library PostDeferredCall function, which schedules a different callback function to be called synchronously.
- 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.
- Call ANSI C functions such as strcpy and sprintf, which affect only the arguments passed in (that is, have no side effects). 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. In your asynchronous callback, 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

OpenDev

`int` **bd = OpenDev** (`char` ***deviceName,** `char` ***instrumentPrefix**);

Purpose

Opens a GPIB device.

Parameters

Input	**deviceName** **instrumentPrefix**	string string	Must be null-terminated. Must be null-terminated.

Return Value

bd	integer	Result of the open device operation.

Return Codes

`-1`	Device table is full, or no more devices available.

Parameter Discussion

deviceName is a string specifying 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`. `OpenDev` uses 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 entered when creating the function tree for the instrument module. If the instrument module has no prefix or if `OpenDev` is not being used in an instrument module, pass the string **`""`** for **instrumentPrefix**.

Using This Function

This function attempts to find an unused device in the GPIB handler's device table and open it. If successful, `OpenDev` returns a device descriptor. Otherwise, it returns a negative number.

ThreadIbcnt

`int` **threadSpecificCount = ThreadIbcnt** (`void`);

Note: ***This function is available only under Windows 95 and NT.***

This function 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 thread-specific) basis. If you are calling GPIB functions in more than one thread, the values in these global variables may not always be reliable.

Status variables analogous to `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained for each thread. This function returns the value of the thread-specific `ibcnt` variable.

If you are not using multiple threads, the value returned by this function is identical to the value of the `ibcnt` global variable.

Parameters

none

Return Value

threadSpecificCount	integer	The 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, this is the error code returned by the MS Windows `LoadLibrary` function.

See Also

`ThreadIbsta, ThreadIberr, ThreadIbcntl.`

ThreadIbcntl

`long` **threadSpecificCount = ThreadIbcntl** (`void`);

Note: ***This function is available only under Windows 95 and NT.***

This function 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 thread-specific) basis. If you are calling GPIB functions in more than one thread, the values in these global variables may not always be reliable.

Status variables analogous to `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained for each thread. This function returns the value of the thread-specific `ibcntl` variable.

If you are not using multiple threads, the value returned by this function is identical to the value of the ibcntl global variable.

Parameters

none

Return Value

threadSpecificCount	long integer	The 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, this is the error code returned by the MS Windows LoadLibrary function.

See Also

ThreadIbsta, ThreadIberr, ThreadIbcnt.

ThreadIberr

int **threadSpecificError = ThreadIberr** (void);

Note: ***This function is available only under Windows 95 and NT.***

This function 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 thread-specific) basis. If you are calling GPIB functions in more than one thread, the values in these global variables may not always be reliable.

Status variables analogous to ibsta, iberr, ibcnt, and ibcntl are maintained for each thread. This function returns the value of the thread-specific iberr variable.

If you are not using multiple threads, the value returned by this function is identical to the value of the iberr global variable.

Parameters

none

Return Value

threadSpecificError	integer	The 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. The system-specific error code is returned by ThreadIbcntl.
ECIC	1	Function requires GPIB-PC to be CIC.
ENOL	2	No listener on write function.
EADR	3	GPIB-PC addressed incorrectly.
EARG	4	Invalid function call argument.
ESAC	5	GPIB-PC not System Controller as required.
EABO	6	I/O operation aborted.
ENEB	7	Non-existent GPIB-PC board.
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 on position.
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 GPIB32.DLL. The MS Windows error code is returned by ThreadIbcntl.
EFNF	203	Unable to find the function in GPIB32.DLL. The MS Windows error code is returned by ThreadIbcntl.
EGLB	205	Unable to find globals in GPIB32.DLL. The MS Windows error code is returned by ThreadIbcntl.
ENNI	206	Not a National Instruments GPIB32.DLL.
EMTX	207	Unable to acquire Mutex for loading DLL. The MS Windows error code is returned by ThreadIbcntl.
EMSG	210	Unable to register callback function with MS Windows.
ECTB	211	The callback table is full.

See Also

`ThreadIbsta, ThreadIbcnt, ThreadIbcntl.`

ThreadIbsta

`int` **threadSpecificStatus = ThreadIbsta** (`void`);

Note: ***This function is available only under Windows 95 and NT.***

This function 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 thread-specific) basis. If you are calling GPIB functions in more than one thread, the values in these global variables may not always be reliable.

Status variables analogous to `ibsta`, `iberr`, `ibcnt`, and `ibcntl` are maintained for each thread. This function returns the value of the thread-specific `ibsta` variable.

If you are not using multiple threads, the value returned by this function is identical to the value of the `ibsta` global variable.

Parameters

none

Return Value

threadSpecificStatus	integer	The 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	Condition
ERR	8000	GPIB error.
END	2000	END or EOS detected.
SRQI	1000	SRQ is on.
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

Chapter 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 on 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 can also be used with a National Instruments RS-485 serial board.

The RS-232 Library Function Panels

The RS-232 Library function panels are grouped in a tree structure according to the types of operations performed. The RS-232 Library function tree appears in Table 5-1.

The first- and second-level bold 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 in plain text are the names of individual function panels. Each RS-232 function panel generates one or more RS-232 function calls. The names of functions are in bold italics to the right of the function panel name.

Table 5-1. The RS-232 Library Function Tree

RS-232	
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***

(continues)

Table 5-1. The RS-232 Library Function Tree (Continued)

Read To File	***ComToFile***
Write Buffer	***ComWrt***
Write Byte	***ComWrtByte***
Write From File	***ComFromFile***
XModem	
XModem Send File	***XModemSend***
XModem Receive File	***XModemReceive***
XModem Configure	***XModemConfig***
Control	
Set Time-out 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 RS232 Error	***ReturnRS232Err***
Get Error String	***GetRS232ErrorString***
Callbacks	
Install COM Callback	***InstallComCallback***

The classes and subclasses in the tree are described below.

- 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 time-out 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 of the functions in the RS-232 Library with the National Instruments RS-485 AT-Serial board. The `ComSetEscape` function 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. A list of the possible error conditions that can occur while using the RS-232 Library functions are at the end of this chapter.

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. Files 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 the `XModemSend` function on the sender side of the transfer and the `XModemReceive` function 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 check sum and retransmit when an error is detected, 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. Basically, 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 section *RS-232 Cable Information* later in this chapter.

All functions, except the `Open` and `Close` functions, require the com port to be opened with `OpenCom` or `OpenComConfig`.

If the program writes data to the output queue and then immediately closes the com port, the data in the queue may be lost if it has not had time to be sent over the port. To guarantee that all bytes were written before closing the port, monitor the length of the output queue with the `GetOutQLen` function. When the output queue length becomes zero, it is safe to close the port.

If the `XModemReceive` function fails to complete properly, verify that the input queue length is greater than or equal to the packet size. Refer to the functions `OpenComConfig` and `XModemConfig`.

If the receiver appears to lose data transmitted by the sender, the input queue of the receiver may be overflowing. This means that the input queue of the receiver is not emptied 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 further 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 are joined with a connector at each end. 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 pins that are numbered and have meaning. The PC pins are numbered and described as 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
20	DTR—Data Terminal Ready *
7	Common

The items with an asterisk (*) indicate the lines that the PC drives, and all other items indicate the lines the PC monitors.

All serial devices are either of the type Data Communication Equipment (DCE) or Data Transmission Equipment (DTE). The PC is of type *DTE*. The difference between the two devices 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 is attached to a DCE device, such as a modem. Therefore, the cable required for a PC (or DTE) to talk to a device that is a DCE is shown in Table 5-3.

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. The cable to connect a PC to a DTE is called a *null modem cable*. A null modem cable must be built as shown in Table 5-4.

Table 5-4. PC to DTE Cable Configuration

(PC)	**Connect pins as indicated:**	**(Device)**
TxD*	2——————3	RxD
RxD	3——————2	TxD*
RTS*	4——————5	CTS
CTS	5——————4	RTS*
DSR	6——————20	DTR
DTR*	20——————6	DSR*
common	7——————7	common

For further 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 can also 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 transmitted by the sender. This condition typically results from the input queue of the receiver not being emptied quickly enough.

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.

Software Handshaking

The `SetXMode` function enables software handshaking. You can use software handshaking when you are transferring ASCII data or text and your serial device 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.

You must not enable software handshaking when transmitting binary data because the special XON/XOFF characters can occur as part of the data stream and are mistaken as control codes. However, you can enable hardware handshaking regardless of the type of data transferred.

No special cable configuration is required to perform software handshaking.

Hardware Handshaking

The `SetCTSMode` function 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 connecting the two devices must include the lines required to support the protocol. Because no single well-defined hardware handshake protocol exists, 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 the DTR line is used to signal its online presence to the other device. Some serial devices also may use the DTR line for hardware handshaking similarly to the CTS line. The `SetCTSMode` function has two different modes to handle either case.

This `SetCTSMode` function 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 RTS and DTR lines are raised when opening the port and lowered when closing the port. Data is sent out the port regardless of the status of CTS.

  Note: ***Under Windows, the*** `SetComEscape` ***function can be used to change the value 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 sending 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 sending 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 used by your device uses the CTS and RTS lines, use a serial cable as shown in Table 5-3 if your device is a DCE, or Table 5-4 if your device is a DTE. Optionally, your cable can omit the connection between pins 6 and 20 if your device does not monitor DSR when sending data. Notice that the RTS pin of the receiver translates to the CTS pin of the sender, and 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 be referencing the pin description in Table 5-2.

RS-232 Library Function Reference

This section describes each function in the LabWindows/CVI RS-232 Library. The LabWindows/CVI RS-232 Library functions are arranged alphabetically.

CloseCom

`int` **result = CloseCom** (`int` **COMPort**);

Purpose

Closes a COM port.

Parameter

Input	**COMPort**	integer	Range 1 through 32.

Return Value

result	integer	Refer to error codes in Table 5-6.

Parameter Discussion

The function does nothing if the port numbers are invalid (port is not open or parameter value is not in the range 1 through 32).

ComBreak

`int` **result = ComBreak** (`int` **COMPort**, `int` **breakTimeMsec**);

Purpose

Generates a break signal.

Parameters

Input	**COMPort** **breakTimeMsec**	integer integer	Range 1 through 32. Range 1 through 255, or 0 to select 250.

Return Value

result	integer	Refer to error codes in Table 5-6.

Using This Function

The function generates a break signal for the number of milliseconds indicated or for 250 ms if the **breakTimeMsec** parameter is zero. For most applications, 250 ms is adequate.

Errors may occur if the port is not open or parameter values are invalid.

ComFromFile

`int` **nbytes = ComFromFile** (`int` **COMPort**, `int` **fileHandle**, `int` **count**,
`int` **terminationByte**);

Purpose

Reads from the specified file and writes to output queue of the specified COM port.

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	fileHandle	integer	File handle returned by `OpenFile`.
	count	integer	If 0, this value is ignored.
	terminationByte	integer	If -1, this value is ignored.

Return Value

nbytes	integer	Number of bytes written to the output queue.
`<0`		Error. Refer to error codes in Table 5-6.

Parameter Discussion

Reads **count** bytes from the file unless it encounters **terminationByte**, reaches EOF, or encounters an error. The function returns the number of bytes successfully written to the output queue. The function returns immediately after placing all bytes in the output queue, not when bytes have all been sent out the com port.

If **count** is zero, the function terminates on **terminationByte**, EOF, or error.

If **terminationByte** is -1, it is ignored, and the function terminates on **count** bytes written, EOF, or error. If **terminationByte** is not -1, reading from the file stops when **terminationByte** is encountered. It does not write **terminationByte** to the output queue. If **terminationByte** is CR or LF, then the function treats CR-LF and LF-CR combinations just as `ComRdTerm` does.

If both **count** and **terminationByte** are disabled, the function terminates on EOF or error.

Using This Function

To guarantee that all bytes were removed from the output queue before closing the port, call `GetOutQLen` to determine the number of bytes remaining in the output queue. If you close the port before every byte has been sent, you lose the bytes remaining in the queue.

The function returns a negative error code if the output queue remains full for the duration of the time-out period, the file handle is bad, a read error occurs, the port is not open, or the **COMPort** is invalid.

ComRd

`int` **nbytes = ComRd** (`int` **COMPort**, `char` **buffer**`[]`, `int` **count**);

Purpose

Reads **count** bytes from input queue of the specified port and stores them in **buffer**. Returns either on time-out or when **count** bytes have been read. Returns an integer value indicating the number of bytes read from queue.

Parameters

Input	**COMPort** **count**	integer integer	Range 1 through 16. 0 value takes no bytes from queue.
Output	**buffer**	string	The buffer in which to store the data.

Return Value

nbytes	integer	Number of bytes read from the input queue.

Using This Function

This function times out if the input queue remains empty in the specified time-out period. This may occur when no data has been received within the time-out period.

The function returns an error code if the port is not open or parameter values are invalid.

Example

```
/* Read 100 bytes from input queue of COM1 into buf. */
int n;
char buf[100];
 :
n = ComRd (1, buf, 100);
if (n != 100)
   /* Time-out or error occurred before read completed. */ ;
```

ComRdByte

`int` **byte = ComRdByte** (`int` **COMPort**);

Purpose

Reads a byte from the input queue of the specified port. Returns an integer whose low-order byte contains the byte read. Returns either on time-out, when the byte is read, or when an error occurs. If an error or a time-out occurs, `ComRdByte` returns a negative error code. See Table 5-6. This is the only case in which the high-order byte of the return value is non-zero.

Parameter

Input	**COMPort**	integer	Range 1 through 32.

Return Value

byte	integer	Low order byte contains the byte read.
`<0`		Error.

Using This Function

This function times out if the input queue remains empty in the specified time-out period. This may occur when no data has been received within the time-out period.

The function returns an error code if the port is not open, **COMPort** is invalid, or a time-out occurs.

ComRdTerm

`int` **nbytes = ComRdTerm** (`int` **COMPort**, `char` **buffer**`[]`, `int` **count**,
`int` **terminationByte**);

Purpose

Reads from input queue until **terminationByte** occurs in **buffer**, **count** is met, or a time-out occurs. Returns integer value indicating number of bytes read from queue.

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	count	integer	If 0, no bytes are removed from queue.
	terminationByte	integer	Low byte contains the numeric equivalent of the terminating character.
Output	**buffer**	string	The buffer in which to store the data.

Return Value

nbytes	integer	Number of bytes read from the input queue.

Using This Function

This function times out if the input queue remains empty within the specified time-out period. This may occur when no data has been received during the time-out period. If the read terminates on the termination byte, the byte is neither written to the buffer nor included in the count.

If the termination character is either 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, discard the LF in addition to the CR.
- If **terminationByte** = LF, and if the character immediately following LF is CR, discard the CR in addition to the LF.

Only the bytes placed in buffer are included in the return count. If CR or LF is discarded because it follows an LF or CR, it is not counted toward satisfying the **count**.

The function returns an error if the port is not open or parameter values are invalid.

ComSetEscape

`int` **result = ComSetEscape** (`int` **COMPort**, `int` **escapeCode**);

Purpose

Directs the specified 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 extended functions are defined by the serial device driver.

Not all device drives support all escape codes. Unknown System Error (-1) is returned when the device driver does not support a particular escape code.

Note: ***This function is supported in the MS Windows version of LabWindows/CVI only.***

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	escapeCode	integer	Specifies the escape code of the extended function.

Return Value

result	integer	Error Code. Refer to Table 5-6.

Parameter Discussion

The following values can be used 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 supported by the system. 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 (data-terminal-ready) signal.

`SETRTS`—Sends the RTS (request-to-send) signal.

`SETXOFF`—Causes the port to act as if an XOFF character has been received.

`SETXON`—Causes the port to act as if an XON character has been received.

The following values may be used only with the RS-485 serial driver developed by National Instruments:

`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 input queue of specified com port and write data to file specified by **fileHandle**. Returns number of bytes successfully written to file. Bytes are read from input queue until **count** is satisfied, **terminationByte** is encountered, or an error occurs, whichever occurs first.

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	fileHandle	integer	File handle returned by `OpenFile`.
	count	integer	If 0, this value is ignored.
	terminationByte	integer	If -1, this value is ignored.

Return Value

nbytes	integer	Number of bytes written to the file.

Parameter Discussion

If **count** is zero, the function ignores it and terminates on **terminationByte** or error.

If **terminationByte** is -1, the function ignores it and terminates on **count** bytes read or an error. If **terminationByte** is valid, the function stops when it encounters **terminationByte**. **terminationByte** is removed from the input queue and is not written to the file. If **terminationByte** is CR or LF, then CR-LF and LF-CR combinations are treated just as they are

for `ComRdTerm`. If both **count** and **terminationByte** are disabled, the function terminates on error (which can include a time-out).

Using This Function

The function returns an error if the output queue remains full for the duration of the time-out period, the file handle is bad, a read error occurs, the port is not open, or the **COMPort** is invalid.

ComWrt

`int` **nbytes = ComWrt** (`int` **COMPort**, `char` **buffer**`[]`, `int` **count**);

Purpose

Writes **count** bytes to the output queue of the specified port. Returns an integer value indicating 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	**COMPort**	integer	Range 1 through 32.
	buffer	string	Buffer containing data to be written, or actual string.
	count	integer	0 value places no bytes in queue.

Return Value

nbytes	integer	Number of bytes placed in the output queue.
`<0`		Error code; See Table 5-6. Byte not placed in the output queue.

Using This Function

This function times out if the output queue has not been updated in the specified time-out period. This can occur if the output queue is full and no further data can be sent because XON/XOFF is enabled and the device has sent an XOFF character without sending the follow-on XON character. It can also occur if Hardware Handshaking is enabled and the Clear To Send (CTS) line is not asserted.

Bytes are sent from the output queue to the serial device under interrupt control without program intervention. If you close the port before all bytes have been sent, you lose the bytes remaining in the queue. To guarantee that all bytes have been removed from the output queue before closing

the port, call `GetOutQLen`. `GetOutQLen` returns the number of bytes remaining in the output queue.

The function returns an error if the port is not open or parameter values are invalid.

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 the specified 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 has failed. Returns immediately without waiting for the byte to be transmitted out through the serial port.

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	byte	integer	Only the low-order byte is significant.

Return Value

status	integer	Result of the write operation.
<0		Error code; See Table 5-6.
1		One byte placed in the output queue.

Parameter Discussion

This function times out if the output queue has not been updated in the specified time-out period. This can occur if the output queue is full and no further data can be sent because XON/XOFF is enabled and the device has sent an XOFF character without sending the follow-on XON character. It can also occur if Hardware Handshaking is enabled and the Clear To Send (CTS) line is not asserted.

Bytes are sent from the output queue to the serial device under interrupt control without program intervention. If you close the port before all bytes have been sent, you lose the bytes remaining in the queue. To guarantee that all bytes have been removed from the output queue before closing the port, call `GetOutQLen`. `GetOutQLen` returns the number of bytes remaining in the output queue.

The function returns an error if the port is not open or parameter values are invalid.

FlushInQ

`int` **status** = **FlushInQ** (`int` **COMPort**);

Purpose

Removes all characters from the input queue of the specified port.

Parameter

Input	**COMPort**	integer	Range 1 through 32.

Return Value

status	integer	Refer to Error Codes in Table 5-6.

Using This Function

You can use this function 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, this function does nothing.

The function returns a negative error code if the port is not open or if **COMPort** is invalid.

FlushOutQ

`int` **status** = **FlushOutQ** (`int` **COMPort**);

Purpose

Removes all characters from the output queue of the specified port.

Parameter

Input	**COMPort**	integer	Range 1 through 32.

Return Value

status	integer	Refer to Error Codes in Table 5-6.

Using This Function

The function returns an error if the port is not open or if **COMPort** is invalid.

GetComStat

`int` **status = GetComStat** (`int` **COMPort**);

Purpose

Returns information about the status of the specified COM port. COM port conditions are accumulated until you call `GetComStat`.

Parameter

Input	**COMPort**	integer	Range 1 through 16.

Return Value

status	integer	Bits indicate COM port status.
<0		Error. Refer to Table 5-5.

Using This Function

Table 5-5 lists definitions of specific bits in the return value. Several bits can be set to 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 (characters were not removed 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.
0080	BREAK	Break signal detected.
1000	REMOTE XOFF	XOFF character received. If XON/XOFF was enabled (see the `SetXMode` function description), no characters are removed from the output queue and sent to the other device until that device sends an XON character.
4000	LOCAL XOFF	XOFF character sent to the other device. If XON/XOFF was enabled (see the `SetXMode` function description), 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 further 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, the application program knows that one or more bytes are invalid. The program cannot know from the status word which byte read since the last call to `GetComStat` is invalid.

The function returns a negative error code if the port is not open or if **COMPort** is invalid.

GetInQLen

`int` **len = GetInQLen** (`int` **COMPort**);

Purpose

Returns the number of characters in the input queue of the specified port. This function does not change the input queue.

Parameter

Input	**COMPort**	integer	Range 1 through 32.

Return Value

len	integer	Number of characters in the input queue.

Parameter Discussion

The function returns an error if the port is not open or if **COMPort** is invalid.

GetOutQLen

`int` **len = GetOutQLen** (`int` **COMPort**);

Purpose

Returns the number of characters in the output queue of the specified port.

Parameter

Input	**COMPort**	integer	Range 1 through 32.

Return Value

len	integer	Number of characters in the output queue.

Using This Function

You can use this function to ensure the output queue has emptied before you close the port. This function has no effect on the output queue.

The function returns an error if the port is not open or if **COMPort** is invalid.

GetRS232ErrorString

`char` ***message** = **GetRS232ErrorString** (`int` **errorNum**)

Purpose

Converts the error number returned by an RS-232 Library function into a meaningful error message.

Parameters

Input	**errorNum**	integer	Error Code returned by RS-232 function.

Return Value

message	string	Explanation of error.

InstallComCallback

`int` **status = InstallComCallback** (`int` **COMPort**, `int` **eventMask**, `int` **notifyCount**,
`int` **eventCharacter**, `ComCallbackPtr` **callbackPtr**,
`void *`**callbackData**);

Note: ***This function is available only in the Windows version of LabWindows/CVI.***

Purpose

This function allows you to install a callback function for a particular COM port. The callback function is called whenever any of the events specified in the **eventMask** parameter occur on the COM port and you allow the system to process events. The system can process events in the following situations.

- You have called `RunUserInterface` and none of your callback functions is currently executing, or
- You call `GetUserEvent`, or
- You call `ProcessSystemEvents`

Only one callback function can apply 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** parameters.

Note: ***The callback function may receive more than one event at a time. When using this function at higher baud rates, some*** `LWRS_RXCHAR` ***events may be missed. It is recommended to use*** `LWRS_RECEIVE` ***or*** `LWRS_RXFLAG` ***instead.***

Note: ***Once the*** `LWRS_RECEIVE` ***event occurs, it is not triggered 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");
}
```

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	eventMask	integer	The events upon which the callback function is called. See the *Parameter Discussion* for a list of valid events. If you want to disable callbacks, pass 0.
	notifyCount	integer	The minimum number of bytes the input queue must contain before sending 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	The name of the user function that processes the event callback.
	callbackData	void *	A pointer to a user-defined four-byte value that is passed to the callback function.

Return Value

status	integer	Refer to error codes in Table 5-6.

Parameter Discussion

The callback function must have the following form.

`void` **CallbackFunctionName** (`int` **COMPort,** `int` **eventMask,** `void *`**callbackData**);

The **eventMask** and **callbackData** parameters are the same values that were passed to `InstallComCallback`.

The events are specified using bits in the **eventMask** parameter. You can specify multiple event bits in the **eventMask** parameter. The valid event bits are listed in the table below.

Bit	Hex Value	Com Port Event	Constant Name
0	0x0001	Any character received.	`LWRS_RXCHAR`
1	0x0002	Received certain character.	`LWRS_RXFLAG`
2	0x0004	Transmit Queue empty.	`LWRS_TXEMPTY`
3	0x0008	CTS changed state.	`LWRS_CTS`
4	0x0010	DSR changed state.	`LWRS_DSR`
5	0x0020	RLSD changed state.	`LWRS_RLSD`
6	0x0040	BREAK received.	`LWRS_BREAK`
7	0x0080	Line status error occurred.	`LWRS_ERR`
8	0x0100	Ring signal detected.	`LWRS_RING`
15	0x8000	**notifyCount** bytes in inqueue.	`LWRS_RECEIVE`

The following table further describes the events.

Event Constant Name	Description
LWRS_RXCHAR	Set when any character is received and placed in the receiving queue.
LWRS_RXFLAG	Set when the event character is received and placed in the receiving queue. The event character is specified in the **eventCharacter** parameter of this function.
LWRS_TXEMPTY	Set when the last character in the transmission queue is sent.
LWRS_CTS	Set when the CTS (clear-to-send) line changes state.
LWRS_DSR	Set when the DSR (data-set-ready) line changes state.
LWRS_RLSD	Set when the RLSD (receive-line-signal-detect) line changes state.
LWRS_BREAK	Set when a break is detected on input.
LWRS_ERR	Set when a line-status error occurs. Line-status errors are CE_FRAME, CE_OVERRUN, and CE_RXPARITY.
LWRS_RING	Set to indicate that a ring indicator was detected.
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.

OpenCom

`int` **result = OpenCom** (`int` **COMPort**, `char` **deviceName**`[]`);

Purpose

Opens a com port.

Parameter

Input	**COMPort**	integer	Range 1 through 32.
	deviceName	string	Name of the COM port.

Return Value

result	integer	Refer to error codes in Table 5-6.

Parameter Discussion

deviceName is the name of the com port in the ASCII string. For example, `COM1` for com port 1 on Microsoft Windows using `COMM.DRV`, and `/dev/ttya` for com port 1 on 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 have specified.

Port Number	deviceName on Windows	deviceName on UNIX
1	"COM1"	"/dev/ttya"
2	"COM2'	"/dev/ttyb"
3	"COM3"	"/dev/ttys1"
4	"COM4"	"/dev/ttys2"
and so on		

Using This Function

`OpenCom` uses 512 bytes of the buffer for the input queue, 512 bytes for the output. The function 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 time-out for I/O operations is 5 seconds. Refer to the functions `SetXMode`, `SetCTSMode`, and `SetComTime` if you want to change these defaults.

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 as specified. If **inputQueueSize** or **outputQueueSize** is between 1 and 29, it is forced to 30.

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	deviceName	string	Name of the COM port.
	baudRate	long	Either 110, 150, 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 56000, 57600, 115200, 128000, or 256000. SPARCstations do not support 14400, 28800, 56000, 57600, 115200, 128000, and 256000. PCs do not support 150. Some PC serial drivers do not support 115200, 128000, and 256000.
	parity	integer	0—no parity. 1—odd parity. 2—even parity. 3—mark parity. 4—space parity.
	dataBits	integer	Either 5, 6, 7, or 8.
	stopBits	integer	Either 1 or 2.
	inputQueueSize	integer	0 selects 512. See discussion below.
	outputQueueSize	integer	0 selects 512. See discussion below.

Return Value

result	integer	Refer to error codes in Table 5-6.

Parameter Discussion

deviceName is the name of the com port in the ASCII string. For example, `COM1` for com port 1 on Microsoft Windows using `COMM.DRV`, and `/dev/ttya` for com port 1 on 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 have specified.

Port Number	deviceName on Windows	deviceName on UNIX
1	"COM1"	"/dev/ttya"
2	"COM2'	"/dev/ttyb"
3	"COM3"	"/dev/ttys1"
4	"COM4"	"/dev/ttys2"
and so on		

Under UNIX, the **inputQueueSize** and **outputQueueSize** parameters are ignored. The serial driver determines the queue size.

Under Windows, if you specify 0 for **inputQueueSize** or **outputQueueSize**, 512 is used. If you specify a value between 0 and 30, 30 is used. On Windows 95 and NT, there is no maximum limitation on the queue size. On Windows 3.1, the maximum queue size is 65535. However, some serial drivers have a maximum of 32767 and give undefined behavior when you use a larger queue size. It is recommended that you use a queue size no greater than 32767.

Under Windows 3.1, the **baudRate** value may be from `0` to `0xffff`. Values below `0xff00` are interpreted by the comm driver literally. Values from `0xff00` to `0xffff` are codes defined by the particular comm driver to represent rates higher than `0xfeff`.

Under Windows 95 and NT, all **baudRate** values are interpreted literally by the comm driver.

Using This Function

The function disables XON/XOFF mode, and CTS hardware handshaking. The default time-out for I/O operations is 5 seconds. Refer to the functions `SetXMode`, `SetCTSMode`, and `SetComTime` if you want to change these defaults.

If the specified port is already open, `OpenComConfig` closes the port (see `CloseCom`) then opens it again.

ReturnRS232Err

`int` **status = ReturnRS232Err** (`void`);

Purpose

Returns the value of `rs232err`.

Parameters

None

Return Value

status	integer	Refer to error codes in Table 5-6.

SetComTime

`int` **result = SetComTime** (`int` **COMPort**, `double` **timeoutSeconds**);

Purpose

Sets time-out limit for input/output operations.

Parameters

Input	**COMPort** **timeoutSeconds**	integer double-precision	Range 1 through 32. Time-out period for all read/write functions.

Return Value

result	integer	Refer to error codes in Table 5-6.

Using This Function

This function sets the time-out parameters for all read and write operations. The default value of **timeoutSeconds** is 5.

For an RS-232 read operation, **timeoutSeconds** specifies the time allowed from the start of the transfer to the arrival of the first byte. It also specifies the time allowed to elapse between the arrival of any two consecutive bytes. An RS-232 read operation waits for at least the specified amount of time for the next incoming byte before it returns a time-out error.

For an RS-232 write operation, **timeoutSeconds** specifies the time allowed before the first byte is transferred to the output queue. It also specifies the time allowed between the transfer of any two consecutive bytes to the output queue. The transfer of bytes to the output queue can become stalled if the output queue is full and hardware or software handshaking is held off. If the hold-off is not resolved within the time-out period, the RS-232 write operation returns a time-out error.

If the **timeoutSeconds** parameter is zero, it disables time-outs and the read or write functions wait indefinitely for the operation to complete.

The function returns an error if the port is not open or parameter values are invalid.

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*.

Parameters

Input	**COMPort** **mode**	integer integer	Range 1 through 32. 0 to disable hardware handshaking, non-zero to enable hardware handshaking. See discussion below.

Return Value

result	integer	Refer to error codes in Table 5-6.

Parameter Discussion

The following are the valid values for **mode**.

0—`LWRS_HWHANDSHAKE_OFF`—Hardware handshaking is disabled. The CTS line is ignored. The RTS and DTR lines are raised the entire time the port is open.

1—`LWRS_HWHANDSHAKE_CTS_RTS_DTR`—Hardware handshaking is enabled. The CTS line is monitored. Both the RTS and DTR lines are used for handshaking.

2—`LWRS_HWHANDSHAKE_CTS_RTS`—Hardware handshaking is enabled. The CTS line is monitored. The RTS is used for handshaking. The DTR line is raised the entire time the port is open.

Using This Function

By default, hardware handshaking is not used.

The function returns an error if the port is not open or parameter values are invalid.

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	**COMPort**	integer	Range 1 through 16.
	mode	integer	0 to disable, non-zero to enable.

Return Value

result	integer	Refer to error codes in Table 5-6.

Using This Function

By default, XON/XOFF sensitivity is disabled. See the *Software Handshaking* section at the beginning of this chapter.

The function returns an error if the port is not open or parameter values are invalid.

XModemConfig

`int` **result = XModemConfig** (`int` **COMPort**, `double` **startDelay**,
`int` **maximum#ofRetries**, `double` **waitPeriod**,
`int` **packetSize**);

Purpose

Sets the XModem configuration parameters for the specified com port.

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	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. >5.0 is recommended.
	packetSize	integer	0 selects the default value 128.

Return Value

result	integer	Result of the XModem configuration operation.
(Less than zero)		Error code; See Table 5-6.
(Zero)		Success.

Parameter Discussion

`XModemSend` and `XModemReceive` use the baud rate, and the input/output queue sizes specified by `OpenComConfig`. But 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 time-out value defined by the `SetComTime` function, XModem functions use a 1 second time-out between data bytes.

A zero input for any parameter except **COMPort** sets that 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. That character is sent 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. Its value must be less than or equal to the input and queue sizes. The standard XModem protocol defines packet sizes to be 128 or 1024. If you are using 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 the com port specified by **COMPort** and writes the packets to the specified file.

Parameters

Input	**COMPort** **fileName**	integer string	Range 1 through 32. Contains the pathname.

Return Value

result	integer	Result of the XModem receive operation.
`<0`		Failure.
`0`		Success.

Using This Function

This function uses the XModem file transfer protocol. The transmitter must also follow this protocol for this function to work properly.

The Xmodem protocol requires that the sender and receiver agree on the error checking protocol. This agreement is negotiated at the beginning of the transfer, and 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.

The file is opened in binary mode, and carriage returns and linefeeds are not treated as ASCII characters. They are written 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 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) bytes of zero. If the packet size is 128, the file contains the five letters in `HELLO` and 123 zero bytes.

The standard XModem protocol only supports 128 and 1024 packet sizes. The sender sends an SOH (0x01) character to indicate that the packet size is 128, or an STX character (0x02) to indicate that the packet size is 1024. LabWindows/CVI attempts to support any packet size. As a receiver, when LabWindows/CVI receives an STX character from the sender, it switches to 1024 packet size regardless of what the user specifies. When it receives an SOH character from the sender, it uses the packet size specified by the user.

For transfers with a large packet size and a low baud rate, a large delay period is recommended.

Example

```
/* Receive the file c:\test\data from COM1 */
/* NOTE:  use \\ in path name in C instead of \. */
int n;
OpenComConfig(1, 9600, 1, 8, 1, 256, 256, 0, 0);
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 **fileName** file and sends it in packets over the com port specified by **COMPort**.

Parameters

Input	**COMPort**	integer	Range 1 through 32.
	fileName	string	Contains the pathname.

Return Value

result <0 0	integer	Result of the XModem send operation. Failure. Success.

Using This Function

The file is opened in binary mode. Carriage returns and linefeeds are not treated as ASCII characters. They are sent to the receiver untouched.

This function uses the XModem file transfer protocol. The receiver must also 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 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) zero bytes.

The standard XModem protocol only supports 128 and 1024 packet sizes. The sender sends an SOH character (0x01) to indicate that the packet size is 128, or an STX character (0x02) to indicate that the packet size is 1024. LabWindows/CVI attempts to support any packet size. As a sender, LabWindows/CVI sends an STX character when you specify packet size as 1024. For any other packet size, it sends an SOH character.

For transfers with a large packet size and a low baud rate, a large delay period is recommended.

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 and the global variable `rs232err` contains that error code. This code is a non-zero value that specifies the type of error that occurred. The currently defined error codes and their meanings are given in Table 5-6.

Table 5-6. RS-232 Library Error Codes

Code	Error Message
-1	Unknown system error.
-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.
-13	Invalid parameter.
-14	Invalid baud rate.
-24	Invalid parity.
-34	Illegal number of data bits.
-44	Illegal number of stop bits.
-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.

(continues)

Table 5-6. RS-232 Library Error Codes (Continued)

-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.
-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 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 CRC.
-508	CRC error.

The value of `rs232err` is zero if the most recently called RS-232 function completed successfully. Errors above 200 occur only on `XModem` function calls. Errors 261 through 265 are recorded when the maximum number of retries has been exhausted in trying to receive an `XModem` function packet.

Chapter 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 Microsoft Windows only.

DDE Library Function Overview

The DDE Library includes functions specifically for Microsoft Windows DDE support. This section contains general information about the DDE Library functions and panels.

The DDE Library Function Panels

The DDE Library function tree appears in Table 6-1. The first- and second-level bold 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 in plain text are the names of individual function panels. Each DDE function panel generates one or more DDE function calls. The names of functions are in bold italics to the right of the function panel name.

Table 6-1. DDE Library Function Tree

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	
Client DDE Execute	***ClientDDEExecute***
Client DDE Read	***ClientDDERead***
Client DDE Write	***ClientDDEWrite***
Connect To DDE Server	***ConnectToDDEServer***
Set Up DDE Hot Link	***SetUpDDEHotLink***
Set Up DDE Warm Link	***SetUpDDEWarmLink***
Terminate DDE Link	***TerminateDDELink***
Disconnect From DDE Server	***DisconnectFromDDEServer***
Get Error String	***GetDDEErrorString***

DDE Clients and Servers

Interprocess communication with DDE involves a client and a server in each DDE conversation. A DDE server can execute commands sent from another application, and send and receive information to and from a client application under Windows. A DDE client can send commands to a server application to be executed, and request data from a server application.

With the LabWindows/CVI DDE Library, you can write programs to act as a DDE client or server. A detailed example using Microsoft Excel and LabWindows/CVI follows later in this chapter to illustrate 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 would like to connect. All DDE server applications have a name and topic that defines the connection. For example, you can connect to Microsoft Excel as a server in two ways with the `ConnectToDDEServer` function from a LabWindows/CVI program. If you want to send commands to be executed by the Excel application, such as opening worksheets and creating charts, 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, where other Windows applications will be sending and receiving commands and data, you need to call the `RegisterDDEServer` function in your program. The `RegisterDDEServer` function establishes your program as a valid DDE server so that other applications can connect to it and exchange information. The server callback function will then be invoked as discussed in the following section.

The 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 object files, a DDE callback function responds to incoming DDE information.

As shown in Table 6-2, a callback function in a client application can respond to only two types of DDE messages: `DDE_DISCONNECT` and `DDE_DATAREADY`. After you set up a warm link or hot link (also called an advisory loop) to another application, the callback function defined in the `SetUpDDEHotLink` or `SetUpDDEWarmLink` function will be called whenever the data values change in the other application, or when the other application is closed.

DDE callback functions used in a program that acts as a DDE server can be triggered in a number of ways from client applications. 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 below:

```
int CallbackFunction (int handle, char *topicName,
                      char *itemName, int xType, int dataFmt,
                      int dataSize, void *dataPtr,
                      void *callbackData);
```

Parameters

Input	**handle**	The conversation handle which uniquely identifies the client server connection.
	topicName	The server application triggering the callback.
	itemName	The 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	The transaction type (see Table 6-2).
	dataFmt	The format of the data being transmitted.
	dataSize	The number of bytes in the data. May actually be greater than the number of bytes transmitted. It is recommended that you encode size information in your data.
	dataPtr	Points to the transmitted data.
	callbackData	A user-defined data value.

Note: ***The value of the dataSize parameter is greater than or equal to the actual size of the data. It is recommended that you encode 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

All of the DDE transaction types (xType) that can trigger a callback function are listed in Table 6-2.

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

Whenever a client program needs to be informed of changes to the value of a particular data item in the server application, a DDE data link is required. You can establish a DDE data link in LabWindows/CVI by calling the `SetUpDDEWarmLink` or `SetUpDDEHotLink` functions. Whenever the data value changes, the client callback function is triggered, and the data is available in the **dataPtr** parameter.

Within one client-server connection, there can be multiple data links, 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. The data item to which the link applies is specified in the **itemName** parameter in the call to `SetUpDDEWarmLink` or `SetUpDDEHotLink` functions.

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, whereas 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.) The `SetUpDDEWarmLink` and `SetUpDDEHotLink` functions are provided because some DDE server applications offer only one type of link.

A 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 can be found in the `samples\ddetcp` directory. The following discussion outlines the process required to open an Excel worksheet file, send data over DDE, and setup 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.

   The function, `ConnectToDDEServer`, with `excel` as the server name and `LWCVI.XLS` as the topic name, establishes a connection with the worksheet. The Callback Function Pointer, `ClientCallback`, identifies the function which will process the DDE transactions generated from this particular conversation.

2. Establishes a DDE warm link with a particular cell in the Excel worksheet.

   The function, `SetUpDDEWarmLink`, with the cell address (`R5C2`) as the item name, establishes a DDE link between the cell in the worksheet. Thereafter, whenever the value of cell `B5` (row 5, column 2) changes, Excel sends information to LabWindows/CVI by triggering the **clientCallbackFunction**.

3. Sends data to the Excel worksheet from LabWindows/CVI.

   After the data is formatted as a string, it is sent to Excel using the `ClientDDEWrite` function with the Excel cell region (`R1C2:R50C2`) as the item name, and the character array, containing 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—The conversation which triggered the callback (multiple DDE conversations can be processed by the same callback function).

   item name—The cell(s) involved.

   topic name—The Excel system or file in Excel involved.

   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 DDE_DATAREADY transaction is received in the callback function, a numeric display is updated by passing the data pointer value to a numeric control on the UIR file. When the DDE event is DDE_DISCONNECT, the DisconnectFromDDEServer function ends the DDE conversation and program execution is halted.

DDE Library Function Reference

AdviseDDEDataReady

```
int status = AdviseDDEDataReady (unsigned int conversationHandle,
                                 char itemName[], unsigned int dataFormat,
                                 void *dataPointer, unsigned int dataSize,
                                 unsigned int timeout);
```

Purpose

Called by a server to write data to a DDE client application. The server should call this only when the value of a data item changes, and a warm or hot link has been established for the data item.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	itemName	string	Uniquely identifies the output item; for example, system.
	dataFormat	unsigned integer	Valid data format; for example, CF_TEXT.
	dataPointer	void pointer	Pointer to buffer holding data.
	dataSize	unsigned integer	Number of bytes in data. Must be 0 if **dataPointer** is NULL. Limited to 64 kbytes under Windows 3.1 and Windows 95.
	timeout	unsigned integer	Timeout in ms.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

dataFormat must be a valid data format recognized by Microsoft Windows. The following are the valid data formats supported by Microsoft Windows:

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

The Microsoft Windows 3.x Programmer's Reference contains an in-depth discussion of DDE programming and meaning of each data format type.

Using This Function

This function allows your program, acting as a DDE server, to send data to a client that has set up a hot or warm link.

When a hot or warm link is set up, your server callback function receives a DDE_ADVISELOOP transaction type (xType) for a particular data object (identified by **itemName**). When the hot or warm link is terminated, your server callback function receives a DDE_ADVISESTOP transaction type 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's value changes, you can call this function, AdviseDDEDataReady, or BroadcastDDEDataReady.

AdviseDDEDataReady differs from BroadcastDDEDataReady in that you specify a particular conversation with a client. AdviseDDEDataReady sends the data only to the specified client, 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 there are other clients with warm links to the same item, they are all notified that new data is available. If they request the new data, your server callback function is invoked with the DDE_REQUESTDATA message. 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 the **dataPointer** and 0 as the **dataSize**, no data is sent to the specified client. Instead, all clients with warm links to the item are notified. If they request the new data, your server callback function is invoked with the DDE_REQUESTDATA message, and you can use the ServerDDEWrite function to send the data in response.

If successful, this function returns the number of bytes sent. Otherwise, this function returns a negative error code. See the help for the **status** control for the error code values.

Note: ***Your program should not call*** `AdviseDDEDataReady` ***in a tight loop because the iterations will compete with user interface events for the CPU time. You should use this function sparingly, and only when the value of the hot- or warm-linked data object changes. In cases when large data objects are to be returned from the server, your program should only call*** `AdviseDDEDataReady` ***when the user interface is not busy.***

See Also

`RegisterDDEServer`, `SetUpDDEHotLink`, `SetUpDDEWarmLink`, `BroadcastDDEDataReady`

BroadcastDDEDataReady

`int` **status = BroadcastDDEDataReady** `(char` **serverName**`[], char` **itemName**`[], char` **topicName**`[], unsigned int` **dataFormat**`, void *`**dataPointer**`, unsigned int` **dataSize**`)`

Purpose

Called by a server to send, to send data to all clients that have set up hot or warm links on the specified topic and item.

Parameters

Input	**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 holding data.
	dataSize	unsigned integer	Number of bytes in data. Limited to 64 KB on Windows 3.1 and Windows 95.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

serverName, topicName, and **itemName** must be strings of length from 1 to 255. They are used without regard to case.

Using this Function

This function allows your program, acting as a DDE server, to send data to all clients that have set up hot or warm links on the specified topic and item.

When a hot or warm link is set up, your server callback function receives a `DDE_ADVISELOOP` transaction type (xType) for a particular data object (identified by **itemName**). When the hot or warm link is terminated, your server callback function receives a `DDE_ADVISESTOP` transaction type 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's value changes, your server program should call either of the following functions, `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, your server callback function is invoked with the `DDE_REQUESTDATA` message. You must call `ServerDDEWrite` in the callback to send the data.

When successful, this function returns the number of bytes sent. Otherwise, this function returns a negative error code. Consult the table at the end of this chapter to see the error code values.

Note: ***Your program should not call this function within a tight loop, because it will compete with user interface events for the CPU time. This function should be used sparingly, and only when the value of the hot or warm linked data object changes. In cases when large data objects are to be returned from the server, it should only be called 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

Called by client to send a command to be executed by a DDE server application.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	commandString	string	Command to be executed by the server application.
	timeout	unsigned integer	Timeout in ms.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

The **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` **status =** **ClientDDERead** (`unsigned int` **conversationHandle**, `char` **itemName**`[]`,
`unsigned int` **dataFormat**, `void` ***dataBuffer**,
`unsigned int` **dataSize**, `unsigned int` **timeout**);

Purpose

Called by client to read data from a DDE server application.

Parameters

Input	**conversationHandle**	unsigned integer	A handle uniquely identifies the conversation.
	itemName	string	Uniquely identifies the output item; for example, `system`.

	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 ms.
Output	**dataBuffer**	void pointer	Buffer in which to receive data.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

dataFormat must be a valid data format recognized by Microsoft Windows. The following are the valid data formats supported by Microsoft Windows:

```
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 programmers' documention for Windows 3.x for an in-depth discussion of DDE programming and meaning of each data format type.

status returns a positive number representing the number of bytes that were successfully read. A negative number corresponds to the error code.

See Also

`ConnectToDDEServer`, `ClientDDEWrite`

ClientDDEWrite

```
int status = ClientDDEWrite (unsigned int conversationHandle, char itemName[],
                             unsigned int dataFormat, void *dataPointer,
                             unsigned int dataSize, unsigned int timeout);
```

Purpose

Called by client to write data to a DDE server application.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	itemName	string	Uniquely identifies the output item; for example, `system`.
	dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
	dataPointer	void pointer	Buffer holding 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 ms.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

dataFormat must be a valid data format recognized by Microsoft Windows. The following are the valid data formats supported by Microsoft Windows:

```
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 programmers' documention for Windows 3.x for an in-depth discussion of DDE programming and meaning of each data format type.

status returns a positive number representing the number of bytes that were successfully read. A negative number corresponds to the 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 (conversation) between your program and a named server on a given topic name.

Parameters

Input	**serverName**	string	Name of the server application.
	topicName	string	Specifies the type of conversation with the server.
	clientCallbackFunction	DDE function pointer	Pointer to the user callback function.
	callbackData	void pointer	User-defined data.
Output	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

The **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. They are used 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 defines a callback function through which all messages from the server will be routed.

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);
```

The **xType** (transaction type) parameter specifies the type of message received from the server.

The **clientCallbackFunction** can receive only two transaction types: `DDE_DISCONNECT` and `DDE_DATAREADY`.

`DDE_DISCONNECT`—Received when a server is requesting the termination of a connection, or when Windows terminates the connection due to 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 data is received in the callback in the **dataPtr** parameter. The **topicName**, **itemName**, **dataFmt**, **dataSize**, and **dataPtr** parameters contain significant data. The **itemName** can specify an object to which the data refers. For example, in Excel, the item name specifies a cell. The **dataFmt** is one of the Windows-defined data types, for example, `CF_TEXT`. The **dataSize** specifies the number of bytes in the data pointed to by **dataPtr**.

Note: ***The* dataSize *value is the value LabWindows/CVI receives from Microsoft Windows. This value can be larger than the actual number of bytes written by the client.***

Note: ***The callback function should return TRUE if the message can be processed successfully. Otherwise, it should return FALSE. The callback function should be short and return as soon as possible.***

callbackData is a four-byte value that will be passed to the callback function each time it is called for this client.

You can 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 would 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.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.

Return Value

status	integer	Refer to error codes in Table 6-3.

Note: ***This function ends a conversation between a client and server corresponding to the* conversationHandle *that was passed. Remember that there can be more than one conversation between a client and a server.***

See Also

`ConnectToDDEServer`, `RegisterDDEServer`

GetDDEErrorString

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

Purpose

Converts the error number returned by a DDE Library function into a meaningful error message.

Parameters

Input	**errorNum**	integer	Status returned by a DDE function.

Return Value

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	**serverName**	string	Name of the server application.
	serverCallbackFunction	DDE function pointer	Pointer to the user callback function.
	callbackData	void pointer	Pointer to the user data.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

serverName must be a string of length from 1 to 255. It is used without regard to case.

The **serverCallbackFunction** is the name of the callback function that will be invoked 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);
```

The **xType** (transaction type) parameter specifies the type of request received from the client. The following transaction types are supported:

`DDE_CONNECT`

```
DDE_DISCONNECT
DDE_DATAREADY
DDE_REQUEST
DDE_ADVISELOOP
DDE_ADVISESTOP
DDE_EXECUTE
```

DDE_CONNECT—This transaction type is received when a client is requesting a connection. The **topicName** parameter specifies the connection topic. The set of valid topic names is defined by the server and can be used in different ways. For example, Excel uses the topic name to specify the file on which the client requests to operate. A client can have multiple connections to the same server as long as there is a different topic name for each connection.

DDE_DISCONNECT—Received when a client is requesting the termination of a connection, or when Windows terminates the connection due to an internal error.

DDE_DATAREADY—Received when the client has sent data via DDE to the server. The **topicName**, **itemName**, **dataFmt**, **dataSize**, and **dataPtr** parameters contain significant data.

The **itemName** can specify an object to which the data refers. For example, in Excel, the item name specifies a cell. The **dataFmt** is one of the Windows-defined data types, for example, CF_TEXT. The **dataSize** specifies the number of bytes in the data pointed to by **dataPtr**.

Note: ***The* dataSize *value is the value LabWindows/CVI receives from Microsoft Windows. This value can be larger than the actual number of bytes written by the client.***

DDE_REQUEST—Received when the client is requesting that data be sent to it via DDE. The **itemName** can specify an object to which the data refers. For example, in Excel, the item name specifies a cell. The **dataFmt** is one of the Windows-defined data types, for example, CF_TEXT.

DDE_ADVISELOOP—Received when the client is requesting a hot or warm link (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 specified item changes value. The server notifies the client of the change in value by calling the function AdviseDDEDataReady or BroadcastDDEDataReady. The **itemName** and **dataFmt** parameters contain significant values. The **itemName** can specify an object to which the data item refers. For example, in Excel, the item name specifies a cell. The **dataFmt** is one of the Windows-defined data types, for example, CF_TEXT.

DDE_ADVISESTOP—Received when the client is requesting the termination of an advisory loop. The **itemName** contains the same value that was used to set up the advisory loop.

DDE_EXECUTE—Received when the client requests the execution of a command. The **itemName** parameter contains the command string. The set of valid command strings is defined by the server. For example, Excel uses "[Save()]" to save a file.

Using This Function

This function registers your program as a DDE server with the specified name. Clients attempting to connect to your program must use the specified name. Thereafter, all requests by the client will be routed through the specified **serverCallbackFunction**.

You can register your program as a DDE server multiple times as long as you specify different server names.

Note: ***The callback function should return TRUE if the request is successful else return FALSE. The callback function should be short and should return as soon as possible***.

callbackData is a four-byte value that will be passed to the callback function each time it is called for this server.

You can define the meaning of the callback data. The following are examples of how the callback data can be used:

1. 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 of the server instances by using the callback data to differentiate between them.

2. You can use the callback data to point to a data object that you need to access in the callback function. In this way, you would 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 status = 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 it requests data.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	itemName	string	Uniquely identifies the output item; for example, `system`.
	dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
	dataPointer	void pointer	Buffer holding 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 ms.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

dataFormat must be a valid data format recognized by Microsoft Windows. The following are the valid data formats supported by Microsoft Windows:

```
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 programmers' documention for Windows 3.x for an in-depth discussion of DDE programming and meaning of each data format type.

Using This Function

This function allows your program, acting as a DDE server, to send data to a client. You should call this function only when your **serverCallbackFunction** receives transaction type (xType) of `DDE_REQUESTDATA`.

If you call the function at any other time, the data is stored until the client requests data. If you call the function multiple times on the same conversation before the client requests the data, each new data set is appended to the buffer containing the stored data.

If the client has set up a hot or warm link and you need to send data *other than* in response to a `DDE_REQUESTDATA` transaction, use the `AdviseDDEDataReady` or `BroadcastDDEDataReady` function.

If successful, this function returns the number of bytes written. Otherwise, this function returns a negative error code.

See Also

`RegisterDDEServer`, `AdviseDDEDataReady`

SetUpDDEHotLink

```
int status = SetUpDDEHotLink (unsigned int conversationHandle, itemName[],
                              unsigned int dataFormat,
                              unsigned int timeout);
```

Purpose

Sets up a hot link (advisory loop) between the client and the server. The function returns zero for success and a negative error code for failure.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	itemName	string	Uniquely identifies the output item; for example, `system`.
	dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
	timeout	unsigned integer	Timeout in ms.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

The **itemName** represents the information in the server application where the DDE link is established. For example, the item name could 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*** **clientCallbackFunction** ***is called with a transaction type of*** `DDE_DATAREADY` ***when the data item is changed at the server site, and 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 only accept 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

Sets up a warm link (advisory loop) between the client and the server. The function returns zero for success and a negative error code for failure.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	itemName	string	Uniquely identifies the output item; for example, `system`.
	dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
	timeout	unsigned integer	Timeout in ms.

Return Value

status	integer	Refer to error codes in Table 6-3.

Parameter Discussion

The **itemName** represents the information in the server application where the DDE link is established. For example, the item name could 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** **clientCallbackFunction** **is called with a transaction type of** `DDE_DATAREADY` **when the data item is changed at the server site**, **and 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 only accept 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 link, previously set up with the server either through `SetUpDDEWarmLink` or `SetUpDDEHotLink`.

This function returns zero for success or a negative error code for failure.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	itemName	string	Uniquely identifies the output item; for example, `system`.
	dataFormat	unsigned integer	Valid data format; for example, `CF_TEXT`.
	timeout	unsigned integer	Timeout in ms.

Return Value

status	integer	Refer to error codes in Table 6-3.

UnregisterDDEServer

`int` **status = UnregisterDDEServer** (`char` **serverName**`[]`);

Purpose

Unregisters your application program as a DDE server.

Parameters

Input	**serverName**	string	Name of the server application.

Return Value

status	integer	Refer to error codes in Table 6-3.

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 non-zero value that specifies the type of error that occurred. Error code return values are negative numbers. The currently defined error codes and their associated meanings are shown in Table 6-3.

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
-6	-kDDE_ReadFailed
-7	-kDDE_WriteFailed
-8	-kDDE_ExecutionFailed
-9	-kDDE_InvalidParameter
-10	-kDDE_OutOfMemory
-11	-kDDE_TimeOutErr
-12	-kDDE_NoConnectionEstablished
-13	-kDDE_FailedToSetUpHotLink
-14	-kDDE_FailedToSetUpWarmLink
-15	-kDDE_GeneralIOErr
-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
-27	-kDDE_PokeAckTimeOut
-28	-kDDE_PostMsgFailed
-29	-kDDE_Reentrancy
-30	-kDDE_ServerDied
-31	-kDDE_SysError
-32	-kDDE_UnadvAckTimeOut
-33	-kDDE_UnfoundQueueId

Note: ***Error codes from -16 to -33 are native DDEML errors which correspond to Windows DDE error codes starting from 0x4000.***

Chapter 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.

In order to use this library in Microsoft Windows, a version of `WINSOCK.DLL` has to be present. 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.

The TCP Library Function Panels

The first- and second-level bold 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 in plain text are the names of individual function panels. Each TCP Library function panel generates one TCP Library function call. The names of the corresponding TCP Library function calls appear in bold italics to the right of the function panel names. The TCP Library function tree appears in Table 7-1.

Table 7-1. The TCP Library Function Tree

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***
Get Error String	***GetTCPErrorString***

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 only request connection 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 followed 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 client. These programs are provided as templates only, and will require modification for operation on your machine.

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 either 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 specifies its own unique port number. These two pieces of information identify different servers either on the same machine or on different machines. Before any client program can connect to a server, it has to know the host name and server port number.

If your program is to act as a TCP server, you must call the `RegisterTCPServer` function in your program. The `RegisterTCPServer` function establishes your program as the server associated with a port number on the local host. Client applications can connect to your program by using either the host name (where the server application is currently running) or the IP address, and the port number associated with the server application. The callback function is invoked whenever the conversation partner requests communication. This is discussed in the following section.

The TCP Callback Function

Callback functions provide the mechanism for receiving notification of connection, connection termination, and data availability. Similar to the method in which 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.

As shown in Table 7-2, a callback function can respond to three types of TCP messages: `TCP_CONNECT`, `TCP_DISCONNECT`, and `TCP_DATAREADY`.

TCP callback functions, used in a program acting as a TCP server, can be triggered in a number of ways from client applications. Whenever a client application attempts to connect to your server program or requests information from your program, the callback function in your

program is invoked to process the request. The parameter prototypes for the TCP callback functions in LabWindows/CVI are defined below:

```
int CallbackFunction (int handle, int xType, int errCode,
                      void *callbackData);
```

where

handle represents the conversation handle
xType represents the transaction type (see table below)
errCode for `TCP_DISCONNECT`, is negative if the connection is being terminated due to an error
callbackData is a user-defined data value.

All of the TCP transaction types (xType) that can trigger a callback function are listed in Table 7-2.

Table 7-2. TCP Transaction Types (xType)

xType	Server	Client	When ?
`TCP_CONNECT`	Y	N	When a new client requests for connection.
`TCP_DISCONNECT`	Y	Y	When conversation partner quits.
`TCP_DATAREADY`	Y	Y	When conversation partner sends data.

Refer to the descriptions for `RegisterTCPServer` and `ConnectToTCPServer` for more information about the TCP callback function.

TCP Library Function Reference

ClientTCPRead

`int` **status = ClientTCPRead** (`unsigned int` **conversationHandle**, `void` ***dataBuffer**, `unsigned int` **dataSize**, `unsigned int` **timeout**);

Purpose

Reads data from a TCP server application when it contains data that is ready for TCP network transmission.

Parameters

Input	**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 ms.

Return Value

status	integer	Returns the number of bytes read, or a negative error code if an error occurs; Refer to error codes in Table 7-3.

See Also

`ConnectToTCPServer`, `ClientTCPWrite`

ClientTCPWrite

`int` **status = ClientTCPWrite** (`unsigned int` **conversationHandle**, `void` ***dataPointer**, `int` **dataSize**, `unsigned int` **timeout**);

Purpose

Writes data to a TCP server application.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	dataPointer	void pointer	Buffer holding data.
	dataSize	unsigned integer	Number of bytes to write.
	timeout	unsigned integer	Timeout in ms.

Return Value

status	integer	Returns the number of bytes written, or a negative error code if an error occurs; Refer to error codes in Table 7-3.

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	**portNumber**	unsigned integer	Uniquely identifies a server on a single machine.
	serverHostName	character array	Can either be the host name or IP address string. For example, `aaa.bbb.ccc` or `123.456.78.90`.
	clientCallbackFunction	TCP function pointer	Pointer to the user callback function.
	callbackData	void pointer	User-defined data.
	timeout	unsigned integer	Timeout in ms.
Output	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.

Return Value

status	integer	Refer to error codes in Table 7-3.

Parameter Discussion

clientCallbackFunction is the name of the function called to process messages to your program as a TCP client.

The callback function must be of the following form:

```
int (*tcpFuncPtr) (int handle, int xType, int errCode, void *callbackData);
```

The **xType** (transaction type) parameter specifies the type of message received from the server. The client callback function can receive the following transaction types.

```
TCP_DISCONNECT

TCP_DATAREADY
```

The **errCode** parameter is used only when the transaction type is `TCP_DISCONNECT`.

The following describes each transaction type.

`TCP_DISCONNECT`—Received when a server is requesting the termination of a connection, or when a connection is being terminated due to an error. If the connection is terminated due to an error, the **errCode** parameter contains a negative error code. Refer to Table 7-3 for the list of error codes.

`TCP_DATATREADY`—Received when the server has sent data via TCP to the client. Your program, acting as the client, should call `ClientTCPRead` to obtain the data.

The client callback function should return TRUE if the message can be processed successfully. Otherwise, the function should return FALSE.

Note: ***The callback function should be short and should return as soon as possible.***

callbackData is a four-byte value that will be passed to the callback function each time it is called for this client.

You should define the meaning of the callback data. One way to use the **callbackData** is as a pointer to a data object that you need to access in the callback function. In this way, you would not need to declare the data object as a global variable.

If you do not want to use the **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.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.

Return Value

status	integer	Refer to error codes in Table 7-3.

Note: ***This function terminates a connection identified by the conversation handle passed. There can be more than one conversation between a client and a server.***

See Also

`ConnectToTCPServer`, `RegisterTCPServer`

DisconnectTCPClient

`int` **status = DisconnectTCPClient** (`unsigned int` **conversationHandle**);

Purpose

Called by a TCP server to terminate a connection with a client. (Be aware that there can be more than one conversation between a server and a client.)

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the connection.

Return Value

status	integer	Refer to error codes in Table 7-3.

See Also

`RegisterTCPServer`

GetTCPErrorString

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

Purpose

Converts the error number returned by a TCP Library function into a meaningful error message.

Parameters

Input	**errorNum**	integer	Status returned by a TCP function.

Return Value

message	string	Explanation of error.

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	**portNumber**	unsigned integer	Uniquely identifies a server on a single machine.
	serverCallbackFunction	TCP function pointer	Pointer to the user callback function.
	callbackData	void pointer	Pointer to the user data.

Return Value

status	integer	Refer to error codes in Table 7-3.

Parameter Discussion

serverCallbackFunction is the name of the function to be called to process client requests.

The callback function must be of the following form:

```
int (*tcpFuncPtr) (int handle, int xType, int errCode,
                   void *callbackData)
```

The **xType** parameter specifies the type of message received from the server. The server callback function can receive the following transaction types.

```
TCP_CONNECT

TCP_DISCONNECT

TCP_DATAREADY
```

The **errCode** parameter is used only when the transaction type is `TCP_DISCONNECT`.

The following describes each transaction type.

`TCP_CONNECT`—The transaction type is received when a client is requesting a connection.

`TCP_DISCONNECT`—Received when a client is requesting the termination of a connection, or when a connection is being terminated due to an error. If the connection is terminated due to an error, the **errCode** parameter contains a negative error code. Refer to Table 7-3 for the list of error codes.

`TCP_DATATREADY`—Received when the client has sent data via TCP to the server. Your program, acting as the server, should call `ServerTCPRead` to obtain the data.

The server callback function should return TRUE if the request is successful. Otherwise, the function should return FALSE.

Note: ***Server callback should be short and should return as soon as possible***.

callbackData is a four-byte value that will be passed to the callback function each time it is called for this server.

It is up to you to define the meaning of the callback data. The following are examples of how the callback data can be used:

- You can register your program as a TCP server multiple times under different port numbers. You could use the same callback function for all of the server instances by using the callback data to differentiate between them.
- You can use the callback data to point to a data object that you need to access in the callback function. In this way, you would 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.

See Also

`ConnectToTCPServer`, `UnregisterTCPServer`

ServerTCPRead

`int` **status =** **ServerTCPRead** (`unsigned int` **conversationHandle**, `void` ***dataBuffer**,
`unsigned int` **dataSize**, `unsigned int` **timeout**);

Purpose

Reads data from a TCP client application.

Parameters

Input	**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 ms.

Return Value

status	integer	Returns the number of bytes written, or a negative error code if an error occurs; Refer to error codes in Table 7-3.

See Also

`RegisterTCPServer`, `ServerTCPWrite`

ServerTCPWrite

`int` **status =** **ServerTCPWrite** (`unsigned int` **conversationHandle**, `void` ***dataPointer**, `unsigned int` **dataSize**, `unsigned int` **timeout**);

Purpose

Writes data to a TCP client application.

Parameters

Input	**conversationHandle**	unsigned integer	Uniquely identifies the conversation.
	dataPointer	void pointer	Buffer holding data.
	dataSize	unsigned integer	Number of bytes to write.
	timeout	unsigned integer	Timeout in ms.

Return Value

status	integer	Returns the number of bytes written, or a negative error code if an error occurs; Refer to error codes in Table 7-3.

See Also

`RegisterTCPServer`, `ServerTCPRead`

UnregisterTCPServer

`int` **status =** **UnregisterTCPServer** (`unsigned int` **portNumber**);

Purpose

Unregisters your server application program as a TCP server.

Parameters

Input	**portNumber**	unsigned integer	Uniquely identifies a server on a single machine.

Return Value

status	integer	Refer to error codes in Table 7-3.

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 non-zero value that specifies the type of error that occurred. Error code return values are negative numbers. The currently defined error codes and their associated meanings are shown in Table 7-3.

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

Chapter 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 Panels* section contains general information about the Utility Library functions and panels. The *Utility Library Function Reference* section contains an alphabetical list of function descriptions.

The Utility Library Function Panels

The Utility Library function panels are grouped in a tree structure according to the type of operations they perform.

The Utility Library function tree is shown in Table 8-1.

The bold headings in the tree are the names of function classes. Function classes are groups of related function panels. The headings in plain text are the names of the individual function panels. The names of the Utility Library functions appear in bold italics beneath the corresponding function panel names.

Table 8-1. The Utility Library Function Tree

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***

(continues)

Table 8-1. The Utility Library Function Tree (Continued)

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***
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***

(continues)

Table 8-1. The Utility Library Function Tree (Continued)

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***
Persistent Variable	
Set Persistent Variable	***SetPersistentVariable***
Get Persistent Variable	***GetPersistentVariable***
Task Switching	
Disable Task Switching	***DisableTaskSwitching***
Enable Task Switching	***EnableTaskSwitching***

(continues)

Table 8-1. The Utility Library Function Tree (Continued)

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***
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***

The classes in the function tree are described here:

- **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 C object modules.
- **Port I/O** functions read and write data from I/O ports (Supported only under Microsoft Windows).

- **Standard Input/Output Window** functions control various attributes of the Standard Input/Output Window.
- **Run-Time Error Reporting** functions enable and disable the feature which breaks execution when a LabWindows/CVI library function returns an error code.
- **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 Microsoft 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 under Microsoft 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 the functions in the LabWindows/CVI Utility Library. The LabWindows/CVI Utility Library functions are arranged alphabetically.

Beep

`void` **Beep** (`void`);

Purpose

Sounds the speaker.

Parameters

None

Return Value

None

Breakpoint

`void` **Breakpoint** (`void`);

Purpose

During execution of a program, a call to `Breakpoint` 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

CloseCVIRTE

`void` **CloseCVIRTE** (`void`)

Purpose

This function releases memory in the LabWindows/CVI Run-Time Engine that was allocated by `InitCVIRTE` 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 after your other detach code.

Parameters

None

Return Value

None

Cls

`void` **Cls** (`void`);

Purpose

In the LabWindows/CVI environment, this function 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	**sourceFileName**	string	File to copy.
	targetFileName	string	Copy of original file.

Return Value

result	integer	Result of copy operation.

Return Codes

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 file names).
-6	Access denied.
-7	Specified path is a directory, not a file.
-8	Disk is full.

Parameter Discussion

sourceFileName and **targetFileName** may contain wildcard characters '?' and '*'. If **sourceFileName** has wildcards, all matching files are copied. If **targetFileName** has wildcards, it will be matched to **sourceFileName**. If the target file is a directory, the existing file (or group of files) will be copied into the directory.

sourceFileName may also be the empty string (**""**), in which case the file found by the most recent call to `GetFirstFile` or `GetNextFile` is copied.

CVILowLevelSupportDriverLoaded

`int` **loaded = CVILowLevelSupportDriverLoaded** (`void`);

Note: ***This function is available only in the Windows 95 and NT version of LabWindows/CVI.***

Purpose

This function returns an indication of whether the LabWindows/CVI low-level support driver was loaded at startup. The following Utility Library functions require the LabWindows/CVI low-level driver to be loaded at startup.

Function	Platforms where low-level support driver is needed
`inp`	Windows NT
`inpw`	Windows NT
`outp`	Windows NT
`outpw`	Windows NT
`ReadFromPhysicalMemory`	Windows 95 and NT
`ReadFromPhysicalMemoryEx`	Windows 95 and NT
`WriteToPhysicalMemory`	Windows 95 and NT
`WriteToPhysicalMemoryEx`	Windows 95 and NT
`DisableInterrupts`	Windows 95
`EnableInterrupts`	Windows 95
`DisableTaskSwitching`	Windows 95

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.

Return Value

loaded	integer	Indicates whether the LabWindows/CVI low-level support driver was loaded at startup.

Return Codes

1	Low-level support driver was loaded at startup.
0	Low-level support driver was not loaded at startup.

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

s	10-character string	The date in MM-DD-YYYY format.

Delay

`void` **Delay** (`double` **numberofSeconds**);

Purpose

Waits the number of seconds indicated by **numberofSeconds**. The resolution on Windows is normally 1 millisecond. However, if the following line appears in the CVI section of your `WIN.INI` file, the resolution is 55 milliseconds.

```
useDefaultTimer = True
```

The resolution on Sun Solaris is 1 millisecond.

Parameter

Input	**numberofSeconds**	double-precision	Number of seconds to wait.

Return Value

None

DeleteDir

`int` **result = DeleteDir** (`char` **directoryName**`[]`);

Purpose

Deletes an existing directory.

Parameters

Input	**directoryName**	String.

Return Value

result	integer	Result of operation.

Return Codes

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	**fileName**	string	File to delete.

Return Value

result	integer	Result of delete operation.

Return Codes

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 may contain wildcard characters '?' and '*' in which case all matching files are deleted.

fileName may also be the empty string (`""`) in which case the file found by the most recent call to `GetFirstFile` or `GetNextFile` is deleted.

DisableBreakOnLibraryErrors

`void` **DisableBreakOnLibraryErrors** (`void`);

Purpose

If debugging is enabled (if the debugging level in the Run Options dialog box of the **Options** menu in the Project window is set to Standard or Extended), this function directs LabWindows/CVI not to display a run-time error dialog box when a National Instruments library function reports an error. If debugging is disabled, this function has no effect.

You can use this function in conjunction with `EnableBreakOnLibraryErrors` to temporarily suppress the Break on Library Errors feature around a segment of code. It does not affect the state of the **Break on Library Errors** check box in the Run Options dialog box of the **Options** menu in the Project window.

Note: ***This function has been superseded by*** `SetBreakOnLibraryErrors`.

DisableInterrupts

`void` **DisableInterrupts** (`void`)**;**

Purpose

Under Windows 3.1 and Windows 95, this function uses the CLI instruction to turn off all maskable 80x86 interrupts. On UNIX, this function uses `sigblock` to block all blockable signals.

Note: ***For you to be able to use this function under Windows 95, the LabWindows/CVI low-level support driver must be loaded.***

Note: ***Under Windows NT, the*** `EnableInterrupts` ***and*** `DisableInterrupts` ***functions have no effect. Interrupts are always enabled while your program is running at the user (as opposed to the kernel) level.***

Parameter

None

Return Value

None

DisableTaskSwitching

`void` **DisableTaskSwitching** (`void`);

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

This function prevents the end-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.

This function affects the behavior of these keys only while LabWindows/CVI or a LabWindows/CVI Standalone Executable is the active application under Microsoft Windows.

This function has no effect in Windows NT. See the *Alternatives in Windows NT* section for instructions on how to achieve the desired effect.

Note: ***To use this function on Windows 95, the LabWindows/CVI low-level support driver must be loaded.***

Disabling the Task List

`DisableTaskSwitching` does not prevent the user from clicking on the desktop to get the task list in Windows 3.1, or clicking on the task bar in 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 will work in Windows 3.1, Windows 95, and Windows NT.

Alternatives in Windows 3.1

Under Windows 3.1, you can prevent the end-user accessing the task list by disabling the Task Manager. Change a 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 must also call `DisableTaskSwitching` to disable the <Alt-Tab> and <Alt-Esc> key combinations. As an alternative to calling `DisableTaskSwitching`, you can arrange for

your standalone application to be brought up in place of the Program Manager when Windows boots. You can do this by changing the following line in your `system.ini [boot]` section.

```
shell = progman.exe
```

to

```
shell = <full-path-of-your-executable>
```

Alternatives in 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.

```
shell = Explorer.exe
```

to

```
shell = <full-path-of-your-executable>
```

Alternatives in 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 following changes to the registry entry for the key name,

```
KEY_LOCAL_MACHINE\Software\Microsoft\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 actions in the following list 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 doing all of the following.

- Call **DisableTaskSwitching** (or use the alternative for Windows NT mentioned in this section).
- Make all of your top-level panels non-movable and non-sizable.

- Do not use the Standard I/O Window in your final application.
- If you use any of the built-in pop-ups in the User Interface Library, make the following calls.

```
SetSystemPopupsAttribute (ATTR_MOVABLE, 0);
SetSystemPopupsAttribute (ATTR_SYSTEM_MENU_VISIBLE, 0);
```

An alternative approach is available on Windows 95 and NT. You can enable timer control callbacks while <Alt-Tab> is pressed, while the system menu is pulled down, or (in some cases) while a window is being moved or sized. You can do this by using the following function call.

```
SetSystemAttribute (ATTR_ALLOW_UNSAFE_TIMER_EVENTS, 1);
```

This alternative is incomplete and can be unsafe. See 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*.

EnableBreakOnLibraryErrors

`void` **EnableBreakOnLibraryErrors** (`void`);

Purpose

If debugging is enabled (if the debugging level in the Run Options dialog box of the **Options** menu in the Project window is set to Standard or Extended), this function directs LabWindows/CVI to display a run-time error dialog box when a National Instruments library function reports an error. If debugging is disabled, this function has no effect.

In general, you should check the **Break on Library Errors** check box in the Run Options dialog box of the **Options** menu in the Project window to enable this feature. However, you can use this function in conjunction with `DisableBreakOnLibraryErrors` to temporarily suppress the Break on Library Errors feature around a segment of code. It does not affect the state of the **Break on Library Errors** check box.

Note: ***This function has been superseded by*** `SetBreakOnLibraryErrors`.

EnableInterrupts

`void` **EnableInterrupts** (`void`);

Under Windows 3.1 and Windows 95, this function uses the STI instruction to turn on all maskable 80x86 interrupts. On UNIX, this function reverses the effect of the last call to `DisableInterrupts`. It restores the signal processing state to the condition prior to the `DisableInterrupts` call.

Note: ***For you to be able to use this function under Windows 95, the LabWindows/CVI low-level support driver must be loaded.***

Note: ***Under Windows NT, the*** `EnableInterrupts` ***and*** `DisableInterrupts` ***functions have no effect. Interrupts are always enabled while your program is running at the user (as opposed to the kernel) level.***

Parameter

None

Return Value

None

EnableTaskSwitching

`void` **EnableTaskSwitching** (`void`);

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

This function lets the user switch to another task by using the <Alt-Tab>, <Alt-Esc>, and <Ctrl-Esc> key combinations, as well as the **Switch-To** item in the **Control/System** menu. This function only affects the behavior of these keys while LabWindows/CVI or a LabWindows/CVI standalone executable is the active application.

ExecutableHasTerminated

`int` **status = ExecutableHasTerminated** (`int` **executableHandle**);

Purpose

Determines whether an application started with `LaunchExecutableEx` has terminated.

Parameters

Input	**executableHandle**	integer	The executable handle acquired from `LaunchExecutableEx`.

Return Value

status	integer	Result of operation.

Return Codes

-1	Handle is invalid.
0	Executable is still running.
1	Executable has been terminated.

Note: ***If you launch another LabWindows/CVI executable under Windows 3.x, the launched executable process will terminate itself after launching the new copy of the CVI Run-time Engine. If you use*** `ExecutableHasTerminated`***, the return value will always be 1 because the process identification for the second Run-time Engine cannot be tracked. See*** `LaunchExecutableEx` ***for more information.***

GetBreakOnLibraryErrors

`int` **state = GetBreakOnLibraryErrors** (`void`);

Purpose

This function returns the state of the **Break on library errors** option. It returns a 1 if the **Break on library errors** option is enabled, or a 0 if it is disabled.

The state of the **Break on Library errors** option can be changed interactively using the **Run Options** command in the **Options** menu of the Project window. The state of the **Break on Library errors** option can also be changed programmatically using `SetBreakOnLibraryErrors`, or the `EnableBreakOnLibraryErrors` and `DisableBreakOnLibraryErrors` functions.

If debugging is disabled, this function always returns 0.

Return Value

state	integer	The current state of the **Break on library errors** option.

Return Codes

`1`	Break on Library Errors option enabled.
`0`	Break on Library Errors option disabled.

GetBreakOnProtectionErrors

`int` **state = GetBreakOnProtectionErrors** (`void`);

Purpose

This function returns the state of the **break on protection errors** feature. It returns a 1 if the option is enabled, or a 0 if it is disabled. If debugging is disabled, this function always returns 0.

For more information on the feature, see the documentation for `SetBreakOnProtectionErrors`.

Return Value

state	integer	The current state of the break on protection errors option.

Return Codes

1	Break on protection errors option enabled.
0	Break on protection errors option disabled.

GetCVIVersion

`int` **versionNum = GetCVIVersion** (`void`);

Purpose

This function 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 that shows in the About LabWindows/CVI dialog box.

For example, for LabWindows/CVI version 4.0, `GetCVIVersion` returns 400. For version 4.1, it would return 410. The values will always increase with each new version of LabWindows/CVI.

The return value of `GetCVIVersion` should not be confused with the predefined macro `_CVI_`, which specifies the version of LabWindows/CVI in which the source file is compiled.

Return Value

versionNum	integer	The version number of LabWindows/CVI or the run-time libraries.

Return Codes

Nnn	Where *N.nn* is the LabWindows/CVI version.

GetCurrentPlatform

`int` **platformCode = GetCurrentPlatform** (`void`);

Purpose

This function returns a code representing the operating system under which a project or standalone executable is running.

The return value of `GetCurrentPlatform` should not be confused with the predefined macros such as `_NI_mswin_`, `_NI_unix_`, and others, which specify the platform on which the project is compiled.

This function is useful when you have a program that can run on multiple operating systems but must take different actions on the different systems. For example, the same standalone executable can run on both Windows 95 and Windows NT. If the program needs to behave differently on the two platforms, you can use `GetCurrentPlatform` to determine the platform at run-time.

Return Value

platformCode	integer	Indicates the current operating system.

Return Codes

`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

GetDir

`int` **result = GetDir** (`char` **currentDirectory**`[]`);

Purpose

Gets the current working directory on the default drive.

Parameter

Output	**currentDirectory**	string	Current directory.

Return Value

result	integer	Result of operation.

Return Codes

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: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Gets the current default drive number and the total number of logical drives in the system.

Parameters

Output	**currentDriveNumber**	integer	Current default drive number.
	numberofDrives	integer	Number of logical drives.

Return Value

result	integer	Result of operation.

Return Codes

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 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 that was loaded using `LoadExternalModule`.

Parameters

Input	**name** **moduleID**	string integer	Name of identifier. ID of loaded module.
Output	**status**	integer	Zero or error code.

Return Value

address	void pointer	Address of the identifier.

Return Codes

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 module.
-25	DLL initialization failed (e.g. DLL file not found).

Parameter Discussion

moduleID is the value `LoadExternalModule` returns**.**

name is the name of the identifier whose address is obtained 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 the function fails, it returns NULL.

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 available on the Windows version of LabWindows/CVI.***

Purpose

Gets the following attributes of a file:

- Read-Only
- System
- Hidden
- Archive

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 **archive** attribute is set whenever you modify the file, and cleared by the DOS BACKUP command.

Parameters

Input	**fileName**	string	File to get attributes.
Output	**read-only**	integer	Read only attribute.
	system	integer	System attribute.
	hidden	integer	Hidden attribute.
	archive	integer	Archive attribute.

Return Value

result	integer	Result of operation.

Return Codes

`0`	Success.
`1`	Specified file is a directory.
`-1`	File not found.

Parameter Discussion

Each attribute parameter will contain one of the following values:

0—attribute is not set

1—attribute is set

fileName may be the empty string (`""`), in which case the attributes of the file found by the most recent call to `GetFirstFile` or `GetNextFile` are returned.

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	**fileName**	string	File to get date.
Output	**month**	integer	Month (1 to 12).
	day	integer	Day of month (1 to 31).
	year	integer	Year (1980–2099).

Return Value

result	integer	Result of operation.

Return Codes

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 may be the empty string (`""`), in which case the date of the file found by the most recent call to `GetFirstFile` or `GetNextFile` is returned (Windows only).

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	**fileName**	string	Name of file.
Output	**fileSize**	long	Size of file in bytes.

Return Value

result	integer	Result of operation.

Return Codes

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 may be the empty string (`""`), in which case the size of the file found by the most recent call to `GetFirstFile` or `GetNextFile` is returned (Windows only).

Example

```
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	**fileName**	string	File to get date.
Output	**hours**	integer	Hours (0 to 23).
	minutes	integer	Minutes (0 to 59).
	seconds	integer	Number of 2-second increments (0-29).

Return Value

result	integer	Result of operation.

Return Codes

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 may be the empty string (`""`), in which case the time of the file found by the most recent call to `GetFirstFile` or `GetNextFile` is returned (Windows only).

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

Starts a search for files with specified attributes and returns the first matching file. If you select multiple attributes, a match occurs on the first file for which one or more of the specified attributes are set and which matches the pattern in the **searchPath** parameter. The search attributes are:

- Normal
- Read-only
- System
- Hidden
- Archive
- Directory

Under UNIX, only the **directory** attribute is honored. 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, all of the attributes are honored. The **normal** attribute specifies files with no other 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 being modified or overwritten. The **system** and **hidden** attributes specify files which normally do not appear in a directory listing. The **directory** attribute specifies directories.

If you pass 1 only for the **normal** attribute, any file that is not read-only, not a system file, not hidden, and not a directory can match. A **normal** file's archive bit may be either on or off. The **normal** attribute is the only attribute that requires other attributes *not* to be set. For example, if you use the **read-only** attribute, any read-only file can match regardless of its other attributes. This holds true for the **system**, **hidden**, **directory**, and **archive** attributes.

If you use more than one attribute, the effect is additive. For example, if you use the **read-only** and **directory** attributes, all read-only files and all directories can match. If you use the **normal** and **read-only** attributes, all normal files and all read-only files can match.

Parameters

Input	**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	**fileName**	string	First file found.

Return Value

result	integer	Result of search.

Return Codes

0	Success.
-1	No files found that match criteria.
-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

searchPath may 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 specified file, if the file is in the currently loaded project.

Parameters

Input	**fileName**	string	Name of file in project.
Output	**fullPathName**	string	Full pathname of file.

Return value

result	integer	Result of operation.

Return codes

0	Success.
-1	File was not found in project.

Parameter Discussion

fileName is the name of a file that is in the currently loaded project. The name must be a simple file name and should not contain any directory paths. For example, `file.c` is a simple file name, whereas `dir\file.c` is not.

fullPathName must be at least `MAX_PATHNAME_LEN` bytes long.

Using This Function

This function 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*:* ***Runtime errors are not reported for this function.***

GetInterruptState

`int` **interruptstate = GetInterruptState** (`void`);

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

This function returns the state of the interrupt bit of the 80x86 CPU status flag.

On Windows NT, this function always returns 1. Interrupts are always enabled while your program is running at the user (as opposed to the kernel) level.

Return Value

interrupt state	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: ***This function only detects keystrokes in the Standard I/O window. It does not detect keystrokes in windows created with the User Interface Library or in the console window in a Windows Console Application.***

Parameters

None

Return Value

k	integer	Key code.

Using This Function

The values returned are the same as the key values used in the User Interface Library. See `userint.h`.

Keystroke	Return Value
<b>	'b'
<Ctrl-b>	(VAL_MENUKEY_MODIFIER | 'B')
<F4>	VAL_F4_VKEY
<Shift-F4>	(VAL_SHIFT_MODIFIER | VAL_F4_VKEY)

Note: ***This function returns -1 if you are running on UNIX and have done one of the following.***

- ***Selected "Use hosts system's standard Input/Output" in the dialog box brought up by selecting* Options » Environment *in the Project window; or***
- ***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** (`char` **directoryName**`[], void *`**moduleHandle**);

Note: ***This function is available only in the Windows 95 and NT versions of LabWindows/CVI.***

Purpose

This function obtains the name of the directory of the specified DLL module.

This function is useful when a DLL and its related files are distributed to multiple users who may 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 the `GetModuleDir` and `MakePathname` functions to construct the full pathname.

If the specified module handle is zero, then this function returns the same result as `GetProjectDir`.

Parameter List

Output	**directoryPathname**	string	Directory of module.
Input	**moduleHandle**	void pointer	Module handle of DLL, or zero for the project.

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**, this function obtains the directory of the project or standalone executable.

Return Value

result	integer	Result of the operation.

Return Codes

0	Success.
-1	The current project has no pathname (that is, it is untitled).
-2	There is no current project.
-3	Out of memory.
-4	The operating system is unable to determine the module directory (**moduleHandle** is probably invalid).

GetNextFile

`int` **result = GetNextFile** (`char` **fileName**`[]`);

Purpose

Gets the next file found in the search starting with `GetFirstFile`.

Parameters

Output	**fileName**	string	Next file found.

Return Value

result	integer	Result of search.

Return Codes

0	Success.
−1	No more files found matching criteria.
−2	`GetFirstFile` must initiate search.

Parameter Discussion

fileName will contain 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 set by `SetPersistentVariable`. However, if you unloaded the project since you last called `SetPersistentVariable`, zero is returned.

In a standalone executable, zero is returned if you have not called `SetPersistentVariable` since the start of execution.

Parameters

Output	**value**	integer	The current value of the persistent variable.

GetProjectDir

`int` **result = GetProjectDir (**`char` **directoryName** `[]`**);**

Purpose

Gets the name of the directory containing the currently loaded project file.

Parameters

Output	**directoryName**	string	Directory of project.

Return value

result	integer	Result of operation.

Return codes

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

This function is useful when a project and its related files are distributed to multiple users who may place them in a different directory on each machine. 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

```
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**);

Purpose

Gets a value indicating the current destination for data written to the standard output (and the source of data read from the standard input.)

The Standard I/O port can be either the CVI Standard Input/Output window or the standard Input/Output of the host system.

This function is valid only on the UNIX version.

Parameters

Output	**stdioPort**	integer	0 = the CVI Standard Input/Output window. 1 = the host system's standard output.

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	**maxNumLines**	integer	The maximum number of lines that can be stored in the Standard Input/Output window. If this amount is exceeded, lines are discarded from the top.
	bringToFrontWhenModified	integer	Indicates whether the Standard Input/Output window is brought to the front each time a string or character is added to it. 1 = Yes. 0 = No.
	showLineNumbers	integer	Indicates whether line numbers are shown in the Standard Input/Output window. 1 = Yes. 0 = No.

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	**top**	integer	The current distance, in pixels, from the top of client area of the Standard Input/Output window to the top of the screen.
	left	integer	The current distance, in pixels, from the leftmost edge of client area of the Standard Input/Output window to the left edge of the screen.

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	**height**	integer	The current height, in pixels, of the client area of the Standard Input/Output window.
	width	integer	The current width, in pixels, of the client area of the Standard Input/Output window.

GetStdioWindowVisibility

`void` **GetStdioWindowVisibility** (`int` ***visible**);

Purpose

Indicates whether the Standard Input/Output window is currently visible. If the window has been made into an icon, it is considered to be *not* visible. If the window cannot be seen merely because its position is off the screen it *is* considered to be visible.

Parameters

Output	**visible**	integer	1 = Standard I/O window is visible. 0 = Standard I/O window is not visible.

GetSystemDate

`int` **status = GetSystemDate** (`int` ***month**, `int` ***day**, `int` ***year**);

Note: ***This function is only available on the Windows version of LabWindows/CVI.***

Purpose

Obtains the system date in numeric format.

Parameters

Output	**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

status	integer	Success or failure.

Return Codes

0	Success.
-1	Failure reported by operating system.

GetSystemTime

`int` **status = GetSystemTime**(`int` ***hours**, `int` ***minutes**, `int` ***seconds**);

Note: ***This function is only available on the Windows version of LabWindows/CVI.***

Purpose

Obtains the system time in numeric format.

Parameters

Output	**hours**	integer	Hours (0–23).
	minutes	integer	Minutes (0–59).
	seconds	integer	Seconds (0–59).

Return Value

status	integer	Success or failure.

Return Codes

0	Success.
-1	Failure reported by operating system.

GetWindowDisplaySetting

`void` **GetWindowDisplaySetting** (`int` ***visible**, `int` ***zoomState**);

Note: ***This function is only available on the Windows version of LabWindows/CVI.***

Purpose

Indicates how the user of your application wants the initial application window to be displayed. The values returned by this function reflect the display options set for the program in Program Manager and other MS Windows shells.

Parameters

Output	**visible**	integer	0, if window is to be hidden; 1, if window is to be displayed.
	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

This function performs initialization of the CVI Run-Time Engine. It is needed only in executables or DLLs that are linked using an external compiler. Otherwise, it is harmless.

Note: ***In LabWindows/CVI version 4.0.1, this function was expanded from one to three parameters. Executables and DLLs created with the one-parameter version of the function will continue to work properly.***

Parameters

Input	**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

status	integer	1 indicates success. 0 indicates failure (probably out of memory).

Using this Function

The function should be called in your `main`, `WinMain`, or `DllMain` function. Which of these three functions you are using determines the parameter values you should 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: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Reads a byte from a port.

Note: ***For you to be able to use this function under Windows NT, the LabWindows/CVI low-level support driver must be loaded.***

Parameters

Input	**portNumber**	integer	The port.

Return Value

byteRead	char	Byte read from port.

---

inpw

`short` **wordRead = inpw** (`int` **portNumber**);

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Reads a word from a port.

Note: ***For you to be able to use this function under Windows NT, the LabWindows/CVI low-level support driver must be loaded.***

Parameters

Input	**portNumber**	integer	The port.

Return Value

wordRead	short	Word read from port.

InStandaloneExecutable

`int` **standalone = InStandaloneExecutable**(`void`);

Purpose

Returns a non-zero value if your program is running as a standalone executable. If your program is running in the LabWindows/CVI development environment, a zero is returned.

Return Value

standalone	integer	1 = Program is running as a standalone executable. 0 = Program is running as in LabWindows/CVI.

KeyHit

`int` **result = KeyHit** (`void`);

Purpose

Indicates whether the user has pressed a key on the keyboard.

Note: ***This function only detects keystrokes in the Standard I/O window. It does not detect keystrokes in windows created with the User Interface Library or in the console window in a Windows Console Application.***

Parameters

None

Return Value

result	integer	Indicates if a key has been pressed.

Return Codes

0	Key has not been pressed.
1	Key has been pressed.

Using This Function

The function 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` will continue to return 1.

Note: ***This function always returns 0 if you are running on UNIX and have done one of the following.***

- ***Selected* Use hosts system's standard Input/Output *in the dialog box brought up by selecting* Options » Environment *in the Project window; or***
- ***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 Microsoft Windows the executable can be either an 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 would like executed. For example, the following command string would copy `file.tmp` from the `temp` directory to the `tmp` directory:

```
command.com /C copy c:\\temp\\file.tmp c:\\tmp
```

Refer to your DOS documentation for further help with `command.com`. DOS executables (`.exe`, `.com`, and `.bat` files) use the settings in `_default.pif` (in your Windows directory) when they are running. You can change their 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.

Parameter

Input	**fileName**	string	Pathname of executable file and arguments.

Return Value

result	integer	Result of operation.

Return Codes Under UNIX

0	Command was successfully started.
-1	The system-imposed limit on the total number of processes under execution or the total number of processes per user would be exceeded. This limit is determined when the system is generated.
-2	There is 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, or the new process image file denies execution permission, or the new process image file is not a regular file.
-5	The length of the path or file, or an element of the environment variable `PATH` prefixed to a file exceeds `{PATH_MAX}`, or a pathname component is longer than `{NAME_MAX}` while `{_POSIX_NO_TRUNC}` is in effect for that file (see 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	The number of bytes used by the new process image's argument list and environment list is greater than `{ARG_MAX}` bytes (see man page for `sysconf(2V)`).
-9	The new process image file has the appropriate access permission, but is not in the proper format.

Return Codes under Microsoft Windows

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.
-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	Attempt was made to load a real-mode application (developed for an earlier version of Windows).
-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. The file must be decompressed before it can be loaded.
-21	Dynamic-link library (DLL) file was invalid. One of the DLLs required to run this application was corrupt.
-22	Application requires Microsoft Windows 32-bit extensions.

Parameter Discussion

fileName is the program to be 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 be passed to the program.

Under Microsoft Windows, if the program is a `.pif`, `.bat`, or `.com` file, the extension must be included in the path name.

For example, under Microsoft Windows the following command string launches the Edit program with the file `file.dat`.

```
c:\\dos\\edit.com c:\\file.dat
```

LaunchExecutableEx

`int` **result = LaunchExecutableEx** (`char` ***fileName**, `int` **windowState**, `int` ***handle**);

Purpose

`LaunchExecutableEx` performs the same operation as `LaunchExecutable` with the following extended features:

- Under Windows, you can specify how the Windows application displays.
- This function returns a handle to the executable that can show whether the executable is still running and also terminate the executable.

Parameters

Input	**fileName**	string	Pathname of executable file and arguments.
	windowState	integer	Specifies how a Windows program is to be shown. (Ignored under UNIX).
Output	**handle**	integer	A handle representing the executable launched.

Return Value

result	integer	Result of operation.

Return Codes

0	Success.
(non-zero value)	Failure (refer to `LaunchExecutable`).

Parameter Discussion

The following values are valid for **windowState**:

`LE_HIDE`	application window is hidden
`LE_SHOWNORMAL`	application window is shown normally and is activated
`LE_SHOWMINIMIZED`	application window is displayed as an icon and is activated
`LE_SHOWMAXIMIZED`	application window is displayed as a maximized window and is activated
`LE_SHOWNA`	application window is shown normally but is not activated
`LE_SHOWMINNOACTIVE`	application window is shown as an icon but is not activated

A handle can be passed to `ExecutableHasTerminated` and `TerminateExecutable`. When you no longer need the handle, you should call `RetireExecutableHandle`. When you do not want to obtain a handle, you can pass NULL.

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 when the processes have already terminated; the program does not recognize that the processes have terminated until you call `RetireExecutableHandle`.

Checking Termination of CVI Executables Under Windows 3.1

If you launch another LabWindows/CVI executable under Windows 3.1, the launched executable process will terminate itself after launching the new copy of the CVI Run-time Engine. If you use `ExecutableHasTerminated`, the return value always will be 1 because the process identification for the second Run-time Engine cannot be tracked. 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. For example:

```
c:\cvi\cvirt4.exe  c:\test\myapp.exe
```

The pathname of the Run-Time Engine might not be `c:\cvi\cvirt4.exe`. You can determine the pathname of the Run-Time Engine by looking at the [`cvirt4`] section in `win.ini`. (If the runtime 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 containing the arguments and pass the pathname of that file as the second argument to the Run-Time Engine. For example:

```
c:\cvi\cvirt4.exe  c:\test\myapp.exe  myargs
```

The file containing the arguments must be in the same directory as the executable. The first three characters in the file containing the arguments must be “CVI” in uppercase, as in the following example:

```
CVI  arg1  arg2  arg3
```

The Run-Time Engine deletes the file containing the arguments after reading it.

LoadExternalModule

`int` **module_id = LoadExternalModule** (`char` **pathName**`[]`);

Purpose

Loads a file containing one or more object modules.

Parameter

Input	**pathName**	string	Relative or absolute pathname of the module to be loaded.

Return Value

module_id	integer	ID of the loaded module.

Return Codes

-1	Out of memory.
-2	File not found.
-4	Invalid file format.
-6	Invalid path name.
-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.
-18	DLL function has an unsupported return type.
-19	A DLL function's argument or return type is a function pointer.
-20	A function in the DLL header file was not found in the DLL.

(continues)

Return Codes (Continued)

-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. (See `LoadExternalModuleEx`.)
-27	Invalid calling module handle. (See `LoadExternalModuleEx`.)
-28	Module loaded in Borland mode in the LabWindows/CVI development environment contains uninitialized global variables that are also defined in other modules.

Parameter Discussion

This function loads an external object module file. The file need not be listed in your project nor loaded as an instrument module.

Under Windows 3.1, the file may be an object file (`.obj`), a library file (`.lib`), or a dynamically linked library (`.dll`). Object and library modules must be compiled with the Watcom C compiler for Windows or the LabWindows/CVI compiler.

Under Windows 95 and NT, the file may be an object file (`.obj`), a library file (`.lib`), or a DLL import library (`.lib`). You cannot load a DLL directly. Object and library modules can be compiled in LabWindows/CVI or an external compiler.

In UNIX, the file may 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 needed and unload them when they are no longer needed.

After a module has been loaded, you can execute its code in one of two ways:

- You can obtain pointers to functions in the module by calling `GetExternalModuleAddr`. You can then call the module's 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. See RunExternalModule for more information.

LoadExternalModule can also be used on a source file (.c) that is part of the current project or a source file that has been loaded 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 have finished testing your module and compiled it into an external object or library file, you need to make no modifications to your application source code other than to change the pathname in the call to LoadExternalModule.

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 containing 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 which can later be passed 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

There is an important difference between loading an object or static library module and loading a DLL via an import library. DLLs are prelinked, that is, when an DLL is loaded, no external references need to be resolved. Object and static library modules, on the other hand, do have external references that need to be resolved. LoadExternalModule resolves them using symbols defined in the project or in object, static library, or import library modules that have already been loaded using LoadExternalModule. This is true even when you call LoadExternalModule from a DLL. LoadExternalModule does not use symbols in a DLL to resolve external references unless those symbols have been exported in the import library.

When you load an object or library module from a DLL, you may want external references to be resolved through global symbols in the DLL that have not been exported in the import library. If this is your intention, you must call LoadExternalModuleEx rather than LoadExternalModule.

Using This Function

pathname may be a relative or absolute pathname. If it is a simple file name (such as module.obj), LoadExternalModule attempts to find the file as follows.

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. If the file has not been found and its extension is `.dll`, `LoadExternalModule` searches for the file in the directories specified in the Windows `LoadLibrary` call.

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.

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

`LoadExternalModuleEx` loads a file containing one or more object modules. It is similar to `LoadExternalModule`, except that, on Windows 95 and NT, external references in object and library modules loaded from a DLL can be resolved using DLL symbols that are not exported. On platforms other than Windows 95 and NT, `LoadExternalModuleEx` works exactly like `LoadExternalModule`.

Parameters

Input	**pathName**	string	Relative or absolute pathname of the module to be loaded.
	callingModuleHandle	void pointer	Usually, the module handle of the calling DLL. You can use `__CVIUserHInst`. Zero indicates the project or executable.

Return Value

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 detailed information on that function.

When you call `LoadExternalModule` on an object or library module, external references need to be resolved. They are resolved using symbols defined in the project or in object, library, or DLL import library modules that have already been loaded using `LoadExternalModule` (or `LoadExternalModuleEx`). This is true even if you call `LoadExternalModule` from a DLL.

You may want to load an object or library module from 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 the **callingModuleHandle** parameter. You can do so by using the LabWindows/CVI pre-defined variable `__CVIUserHInst`.

`LoadExternalModuleEx` first searches the global DLL symbols to resolve external references. Any remaining unresolved references are resolved by searching the symbols defined in the project or in object, library, or import library modules that have already been loaded using `LoadExternalModule` (or `LoadExternalModuleEx`).

`LoadExternalModuleEx` expects the DLL to contain a table of symbols that can be used 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 this table to be included in the DLL. You can do this by creating an include file that includes all of the symbols that need to be in this table. You can then use the **External Compiler Support**

command in the **Build** menu of the Project Window to create an object file containing the table. You must include this object file in the external compiler project you use to create the DLL.

`LoadExternalModuleEx` acts identically to `LoadExternalModule` if either,

- you pass zero for **callingModuleHandle**, or
- you pass `__CVIUserHInst` for **callingModuleHandle,** but you are calling the function from a file that is in the project or your executable, rather than in a DLL, or
- you are not running in Windows 95 or NT.

You cannot load the same external module using two different calling module handles. The function reports an error if you attempt to load the an external module when it is already loaded under a different module handle.

MakeDir

`int` **result = MakeDir** (`char` **directoryName** `[]`);

Purpose

Creates a new directory based on the specified directory name.

Note: ***You can create only one directory at a time.***

Parameters

Input	**directoryName**	string	New directory name.

Return Value

result	integer	Result of operation.

Return Codes

0	Success.
-1	One of the path components 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.
-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 path name from a directory path and a filename. The subroutine ensures that the directory path and the filename are separated by a backslash.

Parameters

Input	**directoryName**	string	Directory path.
	fileName	string	Base file name and extension.
Output	**pathName**	string	Path name.

Return Value

None

Parameter Discussion

pathName must be at least `MAX_PATHNAME_LEN` bytes long. If the **pathName** constructed from **directoryName** and **fileName** exceeds that size, an empty string is returned in **pathName**.

Example

```
char dirname[MAX_PATHNAME_LEN];
char pathname[MAX_PATHNAME_LEN];
GetProjectDir (dirname);
MakePathname (dirname, "FILE.DAT", pathname);
```

---

outp

`char` **byteWritten = outp**(`int` **portNumber**, `char` **byteToWrite**);

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Writes a byte to a port.

Note: ***For you to be able to use this function under Windows NT, the LabWindows/CVI low-level support driver must be loaded.***

Parameters

Input	**portNumber**	integer	The port.
	byteToWrite	char	The byte to be written.

Return Value

byteWritten	char	The byte that was written.

---

outpw

`short` **wordWritten = outpw** (`short` **portNumber,** `int` **wordToWrite);**

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Writes a word to a port.

Note: ***For you to be able to use this function under Windows NT, the LabWindows/CVI low-level support driver must be loaded.***

Parameters

Input	**portNumber**	integer	The port.
	wordToWrite	short	The word to be written.

Return Value

wordWritten	short	The word that was written.

ReadFromPhysicalMemory

```
int status = ReadFromPhysicalMemory (unsigned int physicalAddress,
                                     void *destinationBuffer,
                                     unsigned int numberOfBytes);
```

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Copies the contents of a region of physical memory into **destinationBuffer**. The function does not check whether the memory actually exists. If the memory does not exist, the success value is returned but no data is read.

Note: ***For you to be able to use this function under Windows 95 or NT, the LabWindows/CVI low-level support driver must be loaded.***

Parameters

Input	**physicalAddress**	unsigned integer	The physical address to be read from. There are no restrictions on the address; it can be below or above 1 MB.
	destinationBuffer	void pointer	The buffer into which the physical memory will be copied.
	numberOfBytes	unsigned integer	The number of bytes to copy from physical memory.

Return Value

status	integer	Indicates whether the function succeeded.

Return Codes

1	Success.
0	Failure reported by the operating system, or low-level support driver not loaded.

ReadFromPhysicalMemoryEx

`int` **status** = **ReadFromPhysicalMemoryEx** (`unsigned int` **physicalAddress**,
`void *`**destinationBuffer**,
`unsigned int` **numberOfBytes**,
`int` **bytesAtATime**);

Note: ***This function is available only in the Windows version of LabWindows/CVI.***

Purpose

This function copies the contents of a region of physical memory into the specified buffer. It can copy the data in units of 1, 2, or 4 bytes at a time.

The function does not check whether the memory actually exists. If the memory does not exist, the success value is returned but no data is read.

Note: ***For you to be able to use this function under Windows 95 or NT, the LabWindows/CVI low-level support driver must be loaded.***

Parameters

Input	**physicalAddress**	unsigned integer	The physical address to read from. There are no restrictions on the address; it can be above or below 1 MB.
	destinationBuffer	void pointer	The buffer into which the physical memory is copied.
	numberOfBytes	unsigned integer	The number of bytes to copy from physical memory.
	bytesAtATime	integer	The unit size in which to copy the data. Can be 1, 2, or 4.

Return Value

status	integer	Indicates whether the function succeeded.

Return Codes

1	Success.
0	Failure reported by operating system, or low-level support driver not loaded, or **numberOfBytes** is not a multiple of **bytesAtATime**, or invalid value for **bytesAtATime**.

Parameter Discussion

numberOfBytes must be a multiple of **bytesAtATime**.

ReleaseExternalModule

`int` **status** = **ReleaseExternalModule** (`int` **moduleID**);

Purpose

Decreases the reference count for a module loaded using `LoadExternalModule`.

When `LoadExternalModule` is called successfully on a module, that module's reference count is incremented by one. When you call `ReleaseExternalModule`, its reference count is decremented by one.

If the reference count is decreased to zero, then 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, the external module is removed from memory.

If you want to unload the module regardless of the reference count, call `UnloadExternalModule` rather than `ReleaseExternalModule`. Use `ReleaseExternalModule` when multiple calls may have been made to `LoadExternalModule` on the same module and you do not want to unload the module in case it is still being used by other parts of the application.

Parameter

Input	**moduleID**	integer	The module ID returned by `LoadExternalModule`.

Return Value

status	integer	Indicates the result of the operation.

Return Codes

> 0	Success, but the module was not unloaded. The value indicates the number of remaining references.
0	Success, and the module was unloaded.
-5	The module cannot be unloaded because it is referenced by another external module that is currently loaded.
-9	Invalid module ID.

RenameFile

`int` **result = RenameFile** (`char` **existingFileName**`[]`, `char` **newFileName**`[]`);

Purpose

Renames an existing file.

Parameters

Input	**existingFileName**	string	Existing file name.
	newFileName	string	New file name.

Return Value

result	integer	Result of rename operation.

Return Codes

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 file names).
-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** may contain DOS wildcard characters '?' and '*'. If **existingFileName** has wildcards, all matching files are renamed. If **newFileName** has wildcards, it will be matched to **existingFileName**.

existingFileName may be the empty string (**""**), in which case the file found by the most recent call to `GetFirstFile` or `GetNextFile` is renamed.

Under Microsoft 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 Utility Library that you no longer intend to use the handle acquired from `LaunchExecutableEx`. When you call this function the Utility Library can reuse the memory allocated to keep track of the state of the executable.

Under UNIX, if the process has terminated, the system removes the process from the list of processes. This keeps the system from reaching the limit on the total number of processes under execution by a single user which the system imposes.

Parameters

Input	**executableHandle**	integer	The executable handle acquired from `LaunchExecutableEx`. -1 = handle is invalid. 0 = success.

Return Value

status	integer	Result of 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. This function is encountered in translations.

Parameter

Input	**inputRealNumber**	Double-precision.

Return Value

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 (see `LoadExternalModule`).

Parameters

Input	**moduleID**	integer	ID of loaded module.
	buffer	string	Parameter buffer.

Return Value

result	integer	Indicates the result of the operation.

Return Codes

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 as follows:

```
C:\LW\PROGRAMS\TEST01.OBJ
```

then 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 debugging is enabled and a National Instruments library function reports an error, LabWindows/CVI can display a runtime error dialog box and suspend execution. You can use this function 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 the temporarily disable the **Break on library errors** feature around a segment of code.

This function does not affect the state of the **Break on library errors** checkbox in the **Run Options** command of the Project window.

If debugging is disabled, this function has no effect. Run-time errors are never reported when debugging is disabled.

Parameters

Input	**newState**	integer	Pass a nonzero value to enable. Pass zero to disable.

Return Value

oldState	integer	Previous state of the break on library errors feature.

Return Codes

1	Was previously enabled.
0	Was previously disabled, or debugging is disabled.

Example

```
int oldValue;

oldValue = SetBreakOnLibraryErrors (0);
/* function calls that may legitimately return errors */
SetBreakOnLibraryErrors (oldValue);
```

SetBreakOnProtectionErrors

`int` **oldState = SetBreakOnProtectionErrors** (`int` **newState**);

Purpose

If debugging is enabled, 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 are

- An invalid pointer value is dereferenced in source code.
- An attempt is made in source code to read or write beyond the end of an array.
- A function call is made in source code in which an array is smaller than is expected by the function.
- Pointer arithmetic is performed in source code which generates an invalid address.

You can use this function 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 may want to disable it temporarily around a line of code for which LabWindows/CVI is erroneously reporting a protection error.

If debugging is disabled, this function has no effect. Run-time errors are not reported when debugging is disabled.

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.***

Parameters

Input	**newState**	integer	Pass a nonzero value to enable. Pass zero to disable.

Return Value

oldState	integer	Previous state of the break on protection errors feature.

Return Codes

1	Was previously enabled.
0	Was previously disabled, or debugging is disabled.

Example

```
int oldValue;

oldValue = SetBreakOnProtectionErrors (0);

/* the statement that erroneously reports an error */

SetBreakOnProtectionErrors (oldValue);
```

SetDir

`int` **result = SetDir** (`char` **directoryName**`[]`);

Purpose

Sets the current working directory to the specified directory. Under Windows 3.1, this function can change the current working directory on any drive, however it does not change the default drive. To change the default drive, use the `SetDrive` function.

Parameters

Input	**directoryName**	string	New current working directory.

Return Value

result	integer	Result of operation.

Return Codes

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: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Sets the current default drive.

Parameters

Input	**driveNumber**	integer	New drive number (0 to 25).

Return Value

result	integer	Result of operation.

Return Codes

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: ***This function is available only on the Windows versions of LabWindows/CVI.***

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 **archive** attribute is set whenever the file is modified, and cleared by the DOS BACKUP command.

Parameters

Input	**fileName**	string	File to set attributes.
	read-only	integer	Read-only attribute.
	system	integer	System attribute.
	hidden	integer	Hidden attribute.
	archive	integer	Archive attribute.

Return Value

result	return value	Result of operation.

Return Codes

0	Success.
-1	One of the following errors occurred: • File not found. • 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 may be the empty string (**""**), in which case the attributes of the file found by the most recent call to `GetFirstFile` or `GetNextFile` are set.

SetFileDate

`int` **status = SetFileDate** (`char` **fileName**`[]`, `int` **month**, `int` **day**, `int` **year**);

Purpose

Sets the date of a file.

Parameters

Input	**fileName**	string	File to set attributes.
	month	integer	Month (1 to 12) 1 —January 2 —February 3 —March 4 —April 5 —May 6 —June 7 —July 8 —August 9 —September 10 —October 11 —November 12 —December
	day	integer	Day of month (1 to 31)
	year	integer	Year (1980–2099)

Return Value

status	integer	Result of operation.

Return Codes

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 date, or invalid path (for example, `c:filename` in Windows).
-6	Access denied.

Parameter Discussion

fileName may be the empty string (**""**), in which case the date of the file found by the most recent call to `GetFirstFile` or `GetNextFile` is set.

SetFileTime

`int` **result = SetFileTime** (`char` **fileName**`[]`, `int` **hours**, `int` **minutes**, `int` **seconds**);

Purpose

Sets the time of a file.

Parameters

Input	**fileName**	string	File to set date.
	hours	integer	Hours (0 to 23).
	minutes	integer	Minutes (0 to 59).
	seconds	integer	Seconds (0-58); Odd Values are rounded down.

Return Value

result	integer	Result of operation.

Return Codes

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 time, or invalid path (for example, `c:filename` in Windows).
-6	Access denied.

Parameter Discussion

fileName may be the empty string (`""`), in which case the time of the file found by the most recent call to `GetFirstFile` or `GetNextFile` is set.

seconds value must be entered in increments of 2.

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, the value is reset to zero.

This function 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 be repeated each time you re-run your program. Global variables in your program are reinitialized to zero each time you run your project. Thus, they cannot be used 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. This function sets the value of that variable. To retrieve the variable value, call `GetPersistentVariable()`.

Parameters

Input	**value**	integer	The value to assign to the persistent variable.

SetStdioPort

`int` **status = SetStdioPort** (`int` **stdioPort**);

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 CVI Standard Input/Output window or the standard input/output of the host system.

Note: ***This function is valid only on the UNIX version of LabWindows/CVI.***

Parameters

Input	**stdioPort**	integer	`CVI_STDIO_WINDOW` (0) = the CVI Standard Input/Output window. `HOST_SYSTEM_STDIO` (1) = the host system's standard output.

Return Value

status	integer	Indicates whether the function succeeded.

Return Codes

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 CVI Development System, the default value for **stdioPort** is the current state of the **Use host system's standard input/output** option in the dialog box brought up by the **Environment** command in the **Options** menu of the Project window. The value that you set using this function is reflected the next time you bring up the environment dialog.

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	**maxNumLines**	integer	The maximum number of lines that can be stored in the Standard Input/Output Window. If this amount is exceeded, lines are discarded from the top. Valid range: 100 to 1000000.
	bringToFrontWhenModified	integer	Indicates whether the Standard Input/Output window is brought to the front each time a string or character is added to it. 1 = Yes. 0 = No.
	showLineNumbers	integer	Indicates whether line numbers are shown in the Standard Input/Output window. 1 = Yes. 0 = No.

Return Value

status	integer	Indicates whether the function succeeded.

Return Codes

0	Success.
-1	Maximum number of lines is not within the valid range.

Parameter Discussion

maxNumLines—In an executable, the default value is 10000. In the CVI Development System, the default value is the value set in the dialog box brought up by the **Environment** command in the **Options** menu of the Project window. The value that you set using this function is reflected the next time you bring up the Environment dialog box.

bringToFrontWhenModified—In an executable, the default value is 1 ("bring to front when modified"). In the CVI Development System, the default value is the current state of the "Bring Standard Input/Output window to front whenever modified" option in the dialog box brought up by the Environment command in the **Options** menu of the Project window. The value that you set using this function is reflected the next time you bring up the Environment dialog box.

showLineNumbers—In an executable, the default value is 0 ("do not show line numbers"). In the CVI Development System, the default value is the current state of the **Line Numbers** option in the **View** menu of the Standard Input/Output Window. The value that you set using this function 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	**top**	integer	The 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` -16000 to +16000.
	left	integer	The distance, in pixels, of the leftmost edge of client area of the Standard Input/Output window relative to the leftmost edge of the screen. Valid Range: `VAL_AUTO_CENTER` -16000 to +16000.

Return Value

status	integer	Indicates whether the function succeeded.

Return Codes

`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	**height**	integer	The height, in pixels, of the client area of the Standard Input/Output window. Valid Range: 0 to 16000.
	width	integer	The width, in pixels, of the client area of the Standard Input/Output window. Valid Range: 0 to 16000.

Return Value

status	integer	Indicates whether the function succeeded.

Return Codes

0	Success.
-1	**height** is not within the valid range.
-2	**width** is not within the valid range.

SetStdioWindowVisibility

`void` **SetStdioWindowVisibility** (`int` **visible**);

Purpose

Either brings to the front or hides the Standard Input/Output window.

Parameters

Input	**visible**	integer	1 = Standard I/O window is visible. 0 = Standard I/O window is hidden.

SetSystemDate

`int` **status** = **SetSystemDate** (`int` **month**, `int` **day**, `int` **year**);

Note: ***This function is only available on the Windows version of LabWindows/CVI. Under Windows NT, you must have system administrator status to use this function.***

Purpose

Sets the system date.

Parameters

Input	**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

status	integer	Success or failure.

Return Codes

0	Success.
-1	Failure reported by operating system, probably due to invalid parameter.

SetSystemTime

`int` **status = SetSystemTime**(`int` **hours**, `int` **minutes**, `int` **seconds**);

Note: ***This function is only available on the Windows version of LabWindows/CVI. Under Windows NT, you must have system administrator status to use this function.***

Purpose

Sets the system time.

Parameters

Input	**hours**	integer	Hours (0–23).
	minutes	integer	Minutes (0–59).
	seconds	integer	Seconds (058). Odd values are rounded down.

Return Value

status	integer	Success or failure.

Return Codes

`0`	Success.
`-1`	Failure reported by operating system, probably due to an invalid parameter.

SplitPath

`void` **SplitPath** (`char` **pathName**`[]`, `char` **driveName**`[]`, `char` **directoryName**`[]`, `char` **fileName**`[]`);

Purpose

Splits a path name into the drive name, the directory name, and the file name.

Parameters

Input	**pathName**	string	Path name to be split.
Output	**driveName**	string	Drive name.
	directoryName	string	Full directory path, ending with directory separator character.
	fileName	string	Simple file name.

Return Value

None

Parameter Discussion

The **driveName**, **directoryName**, and **fileName** parameters can each be NULL. If not NULL, they must be buffers of the following size or greater.

drive name	`MAX_DRIVENAME_LEN`
directory name	`MAX_DIRNAME_LEN`
file name	`MAX_FILENAME_LEN`

On operating systems without drive names (such as UNIX), **driveName** will always be filled in 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\"
            file name is          "foo.c"

*/
```

SyncWait

`void` **SyncWait** (`double` **beginTime**, `double` **interval**);

Purpose

Waits until the number of seconds indicated by interval have elapsed since **beginTime**.

Parameters

Input	**beginTime** **interval**	double-precision double-precision	Value returned by `Timer`. Number of seconds to wait after `begin_time`.

Parameter Discussion

beginTime must be a value returned by the `Timer` function.

The resolution on Windows is normally 1 millisecond. However, if the following line appears in the CVI section of your `WIN.INI` file, the resolution is 55 milliseconds.

```
useDefaultTimer = True
```

The resolution on Sun Solaris is 1 millisecond.

Return Value

None

SystemHelp

`int` **status = SystemHelp** (`char` **helpFile**`[]`, `unsigned int` **command**,
`unsigned long` **additionalLongData**,
`char` **additionalStringData**`[]`);

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Starts Windows Help (`WINHELP.EXE`) and passes optional data indicating the nature of the help requested by the application. The application specifies the path of the help file that the application is to display.

For information about creating help files, see the Microsoft Windows Programming Documentation (not included with LabWindows/CVI).

Parameters

Input	**helpFile**	string	Points to a string containing the help file that the Help application is to display.
	command	unsigned integer	Specifies the type of help requested.
	additionalLongData	unsigned long integer	This value parameter depends on the **command** parameter as described in the *Parameter Discussion.*
	additionalStringData	string	This value parameter depends on the **command** parameter as described in the *Parameter Discussion.*

Return Value

status	integer	Non-zero on success, zero on failure.

Parameter Discussion

helpFile contains a filename that may be followed by an angle bracket (>) and the name of a secondary window if the topic is to be displayed in a secondary window rather than in the primary window. The name of the secondary window must have been defined in the [WINDOWS] section of the Help Project (.HPJ) file.

command can be one of the following values:

HELP_COMMAND—Execute a Help Macro. In this case, **additionalStringData** is the Help macro to be executed.

HELP_CONTENTS—Displays the Help contents topic as defined by the Contents option in the [OPTIONS] section of the .HPJ file.

HELP_CONTEXT—Display 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_CONTEXTNOFOCUS—Display Help for a particular topic identified by a context number that has been defined in the [MAP] section of the .HPJ file. Help does not change the focus to the window displaying 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. The main help window is not displayed. In this case, **additionalLongData** is the context number of the topic.

HELP_HELPONHELP—Displays the contents topic of the designated Using Help file.

HELP_KEY—Displays the topic in the keyword list that matches the keyword passed in the **additionalStringData** parameter if there is one exact match. If there is more than one match, it displays the first topic found. If there is no match it displays an error message.

HELP_PARTIALKEY—Displays the topic found in the keyword list that matches the keyword passed in the **additionalStringData** parameter if there is one exact match. If there is more than one match, displays the Search dialog box with the topics listed in the Go To list box. If there is no match, it displays the Search dialog box. If you just want to bring up the Search dialog box without passing a keyword, you should use a pointer to an empty string (**""**).

HELP_POPUPID—Displays in a pop-up window the topic identified by a context string. The main window help is not displayed.

HELP_QUIT—Closes the help file. It will have no effect if the help file was opened by another executable.

HELP_SETCONTENTS—Determines which Contents topic Help should display when the user chooses the Contents button in Help. This call should never be used with HELP_CONTENTS. If a Help file has two or more Contents topics, the application must assign one as the default. To ensure that the correct Contents topic remains set, the application should call SystemHelp() with **command** set to HELP_SETCONTENTS and the **additionalLongData** parameter specifying the corresponding context identifier.

TerminateExecutable

`int` **status = TerminateExecutable** (`int` **executableHandle**);

Purpose

Attempts to terminate an executable if it has not already terminated.

Under Windows the system terminates an executable by sending close messages to each window in the application. If the application does not honor the close messages, then the application does not terminate. The `TerminateExecutable` function gives up control for a limited period to give the application an opportunity to process the close messages. This period should be sufficient for all applications. When you need to allow more time, your program can call the `ProcessSystemEvents` function in a loop, as shown in the following example.

Example

```
#define TIME_LIMIT 5.0 /* number of seconds */
double startTime;
startTime = Timer ();
TerminateExecutable (handle);
while (!ExecutableHasTerminated(handle)
            && (Timer()-startTime > TIME_LIMIT))
      ProcessSystemEvents();
```

Under UNIX, you can allow more time by sending the `SIGKILL` message to the process. The `SIGKILL` message cannot be blocked, caught, or ignored, and therefore should always succeed.

Parameters

Input	**executableHandle**	integer	The executable handle acquired from `LaunchExecutableEx`.

Return Value

status	integer	Result of operation.

Return Codes

-1	Handle is invalid.
0	Handle is invalid.

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 millisecond. However, if the following line appears in the CVI section of your `WIN.INI` file, the resolution is 55 milliseconds.

```
useDefaultTimer = True
```

The resolution on Sun Solaris is 1 millisecond.

Parameters

None

Return Value

t	double-precision	Number of seconds since 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

s	8-character string	The 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.

Parameters

Input	**inputRealNumber**	double-precision.	

Return Value

y	double-precision	Value of **inputRealNumber** without its fractional part.

UnloadExternalModule

`int` **status_id = UnloadExternalModule** (`int` **moduleID**);

Purpose

Unloads an external module file loaded via `LoadExternalModule`.

Parameter

Output	**moduleID**	integer	ID of loaded module.

Return Value

status_id	integer	Indicates the result of the operation.

Return Codes

0	Success.
-9	Failure due to invalid `module_id`.

Parameter Discussion

moduleID is the value returned by `LoadExternalModule`, or -1. If -1 is used, all external modules are unloaded.

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);
}
```

WriteToPhysicalMemory

`int` **status = WriteToPhysicalMemory** (`unsigned int` **physicalAddress**,
`void *`**sourceBuffer**,
`unsigned int` **numberOfBytes**);

Note: ***This function is available only on the Windows versions of LabWindows/CVI.***

Purpose

Copies the contents of **destinationBuffer** into a region of physical memory. The function does not check whether the memory actually exists. If the memory does not exist, the success value is returned but no data is read.

Note: ***For you to be able to use this function under Windows 95 or NT, the LabWindows/CVI low-level support driver must be loaded.***

Parameters

Input	**physicalAddress**	unsigned integer	The physical address to be written to. There are no restrictions on the address; it can be below or above 1 MB.
	sourceBuffer	void pointer	The buffer from which the physical memory will be copied.
	numberOfBytes	unsigned integer	The number of bytes to copy to physical memory.

Return Value

status	integer	Indicates whether the function succeeded.

Return Codes

1	Success.
0	Failure reported by the operating system, or low-level support driver not loaded.

WriteToPhysicalMemoryEx

```
int status = WriteToPhysicalMemoryEx (unsigned int physicalAddress,
                                      void *sourceBuffer,
                                      unsigned int numberOfBytes,
                                      int bytesAtATime);
```

Note: ***This function is available only in the Windows version of LabWindows/CVI.***

Purpose

This function copies the contents of the specified buffer to a region of physical memory. It can copy the data in units of 1, 2, or 4 bytes at a time.

The function does not check whether the memory actually exists. If the memory does not exist, success is returned but no data is written.

Note: ***For you to be able to use this function on Windows 95 or NT, the LabWindows/CVI low-level support driver must be loaded.***

Parameters

Input	**physicalAddress**	unsigned integer	The physical address to write to. There are no restrictions on the address; it can be above or below 1 MB.
	sourceBuffer	void pointer	The buffer from which the physical memory is written.
	numberOfBytes	unsigned integer	The number of bytes to copy to physical memory.
	bytesAtATime	integer	The unit size in which to copy the data. Can be 1, 2, or 4.

Return Value

status	integer	Indicates whether the function succeeded.

Return Codes

1	Success.
0	Failure reported by operating system, or low-level support driver not loaded, or **numberOfBytes** is not a multiple of **bytesAtATime**, or invalid value for **bytesAtATime**.

Parameter Discussion

numberOfBytes must be a multiple of **bytesAtATime**.

Chapter 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.

The *X Property Library Overview* section contains general information about the X Property Library. The *X Property Library Function Reference* section alphabetically lists function names, with descriptions.

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.

The X Property Library Function Panels

The X Property Library function panels are grouped in a tree structure according to the types of operations performed. The X Property Library Function tree appears in Table 9-1.

The first- and second-level bold 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 in plain text are the names of individual function panels. Each X Property Library function panel generates an X Property Library function call. The name of the function is in bold italics to the right of the function panel name.

Table 9-1. The X Property Library Function Tree

Accessing Remote Hosts	
Connect To X Server	***ConnectToXDisplay***
Disconnect From X Server	***DisconnectFromXDisplay***
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***

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 as well as 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 window IDs cannot be known 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 containing the data for transfer to other clients. The other clients read this property from the root window to determine where the data is stored.

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. The function `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, created by the function `CreateXPropType`, 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. See the description of `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.

The Hidden Window

Before you can read or write property data, you need the X window IDs of the windows that will have the properties associated with them.

One option is to always use the root window ID for attaching properties. You could get the root window ID for the local display from the variable `CVIXRootWindow`. To get the root window ID for a remote display you could use the value returned by `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 there is only one root window, there may be conflicts 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 X window ID for this window is available in the X Property Library from the global variable `CVIXHiddenWindow`. 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 of its properties.

Property Callback Functions

You can use the X Property Library to instruct LabWindows/CVI to notify your program whenever a property (or set of properties) on a window (or set of windows) changes. The function `InstallPropertyCallback` registers a function that is called 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 will be either `NewValueX`, if the property value changed, or `DeleteX`, if the property was deleted. The function `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 containing descriptive constant names and numeric values for the errors. `PropLibXErrType` and its enumerated values are all integers. All error values are negative numbers.

The following table lists all the enumerated constant names and their corresponding numeric values. Detailed descriptions of these error types appear in the function descriptions in the following section.

Table 9-3. X Property Library Error Types and Descriptions

Constant Name	Value	Description
NoXErr	0	The function was successful.
InvalidParamXErr	-1	The value passed to one or more of the parameters was invalid. Refer to each function description for specific information.
InvalidDisplayXErr	-2	The **display** argument is not a valid display. The value for this argument must either be the value returned by ConnectToXDisplay or be the predefined value CVIXDisplay.
InvalidWindowXErr	-3	The **window** argument is not a valid window. InstallXPropertyCallback—One or more of the windows in the **windowList** argument are not valid.
InvalidPropertyXErr	-4	The **property** argument is not a valid property handle. This argument must be the value returned by CreateXProperty. InstallXPropertyCallback—One or more of the property handles in the **propertyList** argument are not valid.
InvalidPropTypeXErr	-5	The **propertyType** argument is not a valid property type. This value must either be one of the predefined property types or be a value returned by CreateXPropType.
TooManyConnectionsXErr	-6	The program has already made the maximum number of connections as defined by the constant MAX_X_DISPLAYS. Use DisconnectFromXDisplay to allow more connections.
CannotConnectXErr	-7	The 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.
DupPropertyXErr	-8	A property with the same **propertyName**, but with different **propertyType** already exists.

(continues)

Table 9-3. X Property Library Error Types and Descriptions (Continued)

`DupPropTypeXErr`	-9	A property type with the same **typeName**, but with different **size** or **unit** already exists.
`PropertyInUseXErr`	-10	A property callback was installed with `InstallPropertyCallback` for this property. It is not possible to destroy properties for which callbacks are installed**.**
`PropTypeInUseXErr`	-11	There is a property created by `CreateXProperty` that has this property type. It is not possible to destroy property types if there are properties that use them**.**
`TypeMismatchXErr`	-12	The actual X type of the property value on the window does not match the type specified for **property**.
`UnitMismatchXErr`	-13	The actual X format of the property value on the window does not match the unit specified for **property**.
`InvalidIndexXErr`	-14	The **index** specified is larger than the actual number of property items on the window.
`SizeMismatchXErr`	-15	The number of bytes in the property value is not a multiple of the size specified for **property**.
`OverflowXErr`	-16	Arithmetic overflow occurred with calculations involving the property item sizes and the number of items specified.
`InvalidCallbackXErr`	-17	The function specified is not installed as a callback.
`MissingPropertyXErr`	-18	The property does not exist on the window.
`InsuffMemXErr`	-19	There is insufficient memory to perform the operation. `CreateXProperty`—There is insufficient memory to store the property information or there are already 256 properties. `CreateXPropType`—There is insufficient memory to store the property information or there are already 64 property types.
`GeneralXErr`	-20	An Xlib function failed for an unknown reason.
`BrokenConnectionXErr`	-21	The connection to the X server was broken. This occurs if the remote server terminated.

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 functions of the X Property Library 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`, `CVIXHiddenWindow` of the X Property Library.

- `void` **HandlePropertyNotifyEvent**(`EventPtrX` **event**)

  This function calls the functions that are installed as property callbacks. You should call this function whenever you receive an `XPropertyNotify` event to automatically invoke callback functions. The event must be a valid `XPropertyEvent`.

X Property Library Function Reference

This section describes the functions in the LabWindows/CVI X Property Library. The LabWindows/CVI X Property functions are arranged alphabetically.

ConnectToXDisplay

`PropLibXErrType` **status = ConnectToXDisplay** (`const char` ***displayName**, `DisplayPtrX` ***display**, `WindowX` ***rootWindow**);

Purpose

Connect to a remote X server.

Use this function 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 using 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 used by LabWindows/CVI.

Parameters

Input	**displayName**	string	Determines the X server connection and which communication domain to use.
Output	**display**	`DisplayPtrX` (passed by reference)	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` (passed by reference)	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

The return value indicates the success or failure status of the function call. A negative value indicates an error. **status** values are shown in the following table.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to one or more of the parameters.
`TooManyConnectionsXErr`	-6	The program has already made the maximum number of connections as defined by the constant `MAX_X_DISPLAYS`. Use `DisconnectFromXDisplay` to allow more connections.
`CannotConnectXErr`	-7	The 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 on which the display is physically connected.
- `server` specifies the number of the server on its host computer (usually 0).
- `screen` specifies the number of the default screen on the server (usually 0).

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about the `XOpenDisplay` and `DefaultRootWindow` functions.

CreateXProperty

```
PropLibXErrType status = CreateXProperty (const char *propertyName,
                                          PropTypeHandleX propertyType,
                                          PropertyHandleX *property);
```

Purpose

Create 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 this function 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	**propertyName**	string	Name of the property. Each property name is unique and has a type, which cannot be changed once the property is created.
	propertyType	`PropTypeHandleX`	Type of the property. This value must be either a predefined type or a value returned by `CreateXPropType`.
Output	**property**	`PropertyHandleX` (passed by reference)	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

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table **shows status values.**

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to one or more of the parameters.
`InvalidPropTypeXErr`	-5	The **propertyType** argument is not a valid property type. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.
`DupPropertyXErr`	-8	A property with the same **propertyName**, but with different **propertyType** already exists.
`InsuffMemXErr`	-19	There is insufficient memory to store the property information or there are already 256 properties.

Parameter Discussion

propertyType is added with 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 the `XInternAtom` function.

CreateXPropType

`PropLibXErrType` **status = CreateXPropType** (`const char` ***typeName**,
`unsigned int` **size**, `unsigned int` **unit**,
`PropTypeHandleX` ***propertyType**);

Purpose

Creates 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 cannot be changed except by destroying the property and recreating it.

There are three predefined property types that you do not need to create using this function. These types, listed below, are useful for defining properties to store window IDs, integers and strings.

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)`

Note: ***You can create a maximum of 64 different property types.***

Parameters

Input	**typeName**	string	Name of the property type. Each property type name is unique and has one set of attributes, which cannot be changed 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	**propertyType**	`PropTypeHandleX` (passed by reference)	Property type created. Use this value as the type parameter to `CreateXProperty` to create properties.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. **status** values are shown in the following table.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to one or more of the parameters; **size** argument is 0; **unit** is not 1, 2, or 4; or **size** is not a multiple of **unit**.
`DupPropTypeXErr`	-9	A property type with the same **typeName**, but with different **size** or **unit** already exists.
`InsuffMemXErr`	-19	There is insufficient memory to store the property information or there are already 64 property types.

Parameter Discussion

Usually, you can use the expression `sizeof(`*`TYPE`*`)` for the **size** parameter, where *`TYPE`* is the C data type (`char`, `int`, and others) used to store the property value. This value must be a multiple of the **unit** argument.

unit specifies how the X server should view the property item (as an array of 1-byte, 2-byte or 4-byte objects) and is necessary to perform simple byte-swapping between different types of computers. See the notes near the end of this function description.

If the property item consists of a single object, such as an integer or a character, the unit should be just 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 containing 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 has different byte-ordering than your application, the unit specified is used to perform the byte swapping. However, byte swapping cannot be properly performed for structures containing different size members or for*** `double` ***type. For these special cases, use a unit of 1 and then explicitly perform byte swapping where needed.***

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 used by Xlib functions 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 the `XInternAtom` function.

DestroyXProperty

`PropLibXErrType` **status = DestroyXProperty** (`PropertyHandleX` **property**);

Purpose

Destroys X property information. You can use this function when you no longer need to access a property. This function frees memory allocated by `CreateXProperty`. The property handle cannot be used after this function is called.

All property information is destroyed when the program terminates.

Note: ***It is not possible to destroy properties for which callbacks are installed.***

Parameter

Input	**property**	`PropertyHandleX`	Handle to the property information to be destroyed. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidPropertyXErr`	-4	The **property** argument is not a valid property. This argument must be the value returned by `CreateXProperty`.
`PropertyInUseXErr`	-10	A property callback was installed with `InstallPropertyCallback` for this property.

DestroyXPropType

`PropLibXErrType` **status = DestroyXPropType** (`PropTypeHandleX` **propertyType**);

Purpose

Destroys X property type. You can use this function when you no longer need a property type. This function frees memory that was allocated by `CreateXPropType`. The property type cannot be used after this function is called.

All property types are destroyed when the program terminates.

Note: ***It is not possible to destroy property types if there are properties that use them.***

Parameter

Input	**propertyType**	`PropertyHandleX`	Handle of the property type to be destroyed. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidPropTypeXErr`	-5	The **propertyType** argument is not a valid property type. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.
`PropTypeInUseXErr`	-11	There is a property created by `CreateXProperty` that has this property type.

DisconnectFromXDisplay

`PropLibXErrType` **status = DisconnectFromXDisplay** (`DisplayPtrX` **display**);

Purpose

Disconnects from a remote X server. You can use this function to end access to a remote X server you connected using `ConnectToXDisplay`. After this function is called, you can no longer access the remote X server.

Parameter

Input	**display**	`DisplayPtrX`	A pointer to the display of the remote X server to be disconnected. This value must have been obtained from `ConnectToXDisplay`.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to the parameter.
`InvalidDisplayXErr`	-2	The **display** argument is not a valid display. This value must be the value returned by `ConnectToXDisplay`.

See Also

Refer to the *Xlib Programming Manual* or to *Xlib—C Language X Interface, MIT X Consortium Standard* for more information about the `XCloseDisplay` function.

GetXPropErrorString

`char` ***message = GetXPropErrorString** (`PropLibXErrType` **errorNum**)

Purpose

Converts the error number returned by an X Property Library function into a meaningful error message.

Parameters

Input	**errorNum**	`PropLibXErrType`	Status returned by an X Property function.

Return Value

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	**property**	`PropertyHandleX`	Property handle for which the name is to be obtained. This value must have been obtained from `CreateXProperty`.
Output	**propertyName**	character pointer (passed by reference)	Pointer to the property name.

Warning: ***The* propertyName *pointer points to memory allocated by*** `CreateXProperty`***. You must not attempt to free this pointer or to change its contents.***

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to the name parameter.
`InvalidPropertyXErr`	-4	The **property** argument is not a valid property handle. This argument must be the value returned by `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	**property**	`PropertyHandleX`	Property handle for which the name is to be obtained. This value must have been obtained from `CreateXProperty`.
Output	**propertyType**	`PropTypeHandleX` (passed by reference)	The property type. Use the functions `GetXPropTypeName`, `GetXPropTypeSize`, and `GetXPropTypeUnit` to get more information about the property type.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to the parameter.
`InvalidPropertyXErr`	-4	The **property** argument is not a valid property handle. This argument must be the value returned by `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	**propertyType**	`PropTypeHandleX`	Handle to property type for which the name is to be obtained. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.
Output	**typeName**	character pointer (passed by reference)	The property type name.

Warning: ***The* typeName *pointer points to memory allocated by*** `CreateXPropType`***. You must not attempt to free this pointer or to change its contents.***

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to the name parameter.
`InvalidPropTypeXErr`	-5	The **propertyType** argument is not a valid property type. This value must either be one of the predefined property types or be a value returned by `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. The size is the number of bytes in a single property item.

Parameters

Input	**propertyType**	`PropTypeHandleX`	Handle to property type for which the size is to be obtained. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.
Output	**size**	unsigned integer (passed by reference)	The size associated with the property type. The size is the number of bytes in a single property item.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to the **size** parameter.
`InvalidPropTypeXErr`	-5	The **propertyType** argument is not a valid property type. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.

See Also

`CreateXPropType`

GetXPropTypeUnit

`PropLibXErrType` **status = GetXPropTypeUnit** (`PropTypeHandleX` **propertyType**,
`unsigned int` ***unit**);

Purpose

Get a property type unit.

This function returns the 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.

Parameters

Input	**propertyType**	`PropTypeHandleX`	Handle to property type for which the unit is to be obtained. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.
Output	**unit**	unsigned integer (passed by reference)	The **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

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to the **unit** parameter.
`InvalidPropTypeXErr`	-5	The **propertyType** argument is not a valid property type. This value must either be one of the predefined property types or be a value returned by `CreateXPropType`.

See Also

`CreateXPropType`

GetXWindowPropertyItem

```
PropLibXErrType status = GetXWindowPropertyItem (DisplayPtrX display,
                                                 WindowX window,
                                                 PropertyHandleX property,
                                                 void *propertyItem);
```

Purpose

Get a single property item from a window.

This function obtains the value of the specified property on the window and copies a single item into the supplied buffer. When there are more than one item in the property value, this function obtains only the first one. This function does not change the property value.

If the property does not exist on the window, this function reports the `MissingPropertyXErr` error.

Use the function `GetXWindowPropertyValue` to get multiple property items.

Parameters

Input	**display**	`DisplayPtrX`	A pointer to the display of the X server to which the window belongs.
	window	`WindowX`	The window from which the property item is to be obtained.
	property	`PropertyHandleX`	Handle of the property to be obtained. This value must have been obtained with `CreateXProperty`.
Output	**propertyItem**	generic pointer	Property item obtained from window.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

NoXErr	0	The function was successful.
InvalidParamXErr	-1	NULL was passed to one or more parameters.
InvalidDisplayXErr	-2	The **display** argument is not a valid display. This argument must either be the predefined value CVIXDisplay or be the value returned by ConnectToXDisplay.
InvalidWindowXErr	-3	The **window** argument is not a valid window.
InvalidPropertyXErr	-4	The **property** argument is not a valid property handle. This argument must be the value returned by CreateXProperty.
TypeMismatchXErr	-12	The actual X type of the property value on the window does not match the type specified for **property**.
UnitMismatchXErr	-13	The actual X format of the property value on the window does not match the unit specified for **property**.
SizeMismatchXErr	-15	The number of bytes in the property value is not a multiple of the size specified for **property**.
MissingPropertyXErr	-18	The property does not exist on the window.
InsuffMemXErr	-19	There is insufficient memory to perform the operation.
GeneralXErr	-20	An Xlib function failed for some unknown reason.
BrokenConnectionXErr	-21	The connection to the X server was broken. This occurs if the remote server terminated.

Parameter Discussion

display must either be the predefined value CVIXDisplay or be the value returned by ConnectToXDisplay. Use CVIXDisplay if the window is on the same display used by LabWindows/CVI.

For the **window** parameter, use CVIXRootWindow to access the default root window of the display used by LabWindows/CVI. 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 the `XGetWindowProperty` function.

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

Get the value of a property on a window.

This function 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, this function does NOT report an error. Instead, the number of items returned is set to 0.***

Parameters

Input	**display**	`DisplayPtrX`	A pointer to the display of the X server to which the window belongs.
	window	`WindowX`	The window from which the property value is to be obtained.
	property	`PropertyHandleX`	Handle of the property to be obtained. This value must have been obtained with `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.
	numberofItemsRequested	unsigned integer	Number of property items to obtain from the window.
	delete	integer	Flag indicating 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	**numberofItemsReturned**	unsigned integer (passed by reference)	Number of property items that were obtained from the window.
	numberOfItemsRemaining	unsigned integer (passed by reference)	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

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to one or more parameters.
`InvalidDisplayXErr`	-2	The **display** argument is not a valid display. This argument must either be the predefined value `CVIXDisplay` or be the value returned by `ConnectToXDisplay`.
`InvalidWindowXErr`	-3	The **window** argument is not a valid window.
`InvalidPropertyError`	-4	The **property** argument is not a valid property handle. This argument must be the value returned by `CreateXProperty`.
`TypeMismatchXErr`	-12	The actual X type of the property value on the window does not match the type specified for **property**.
`UnitMismatchXErr`	-13	The actual X format of the property value on the window does not match the unit specified for **property**.
`InvalidIndexXErr`	-14	The **index** specified is larger than the actual number of property items on the window.
`SizeMismatchXErr`	-15	The number of bytes in the property value is not a multiple of the size specified for **property**.
`InsuffMemXErr`	-19	There is insufficient memory to perform the operation.
`GeneralXErr`	-20	An Xlib function failed for some unknown reason.
`BrokenConnectionXErr`	-21	The connection to the X server was broken. This occurs if the remote server terminated.

Parameter Discussion

display must either be the predefined value `CVIXDisplay` or be the value returned by `ConnectToXDisplay`. Use `CVIXDisplay` if the window is on the same display used by LabWindows/CVI.

For the **window** parameter, use `CVIXRootWindow` to access the default root window of the display used by LabWindows/CVI. Use `CVIXHiddenWindow` to access the hidden window associated with your application.

numberofItemsReturned will be less than or equal to the number of property items requested. If the property does not exist on the window or there is no property value, this value will be 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 the `XGetWindowProperty` function.

InstallXPropertyCallback

```
PropLibXErrType status = InstallXPropertyCallback (DisplayPtrX display,
                         const WindowX windowList[],
                         unsigned int numberofWindows,
                         const PropertyHandleX propertyList[], unsigned
                         int numberofProperties,
                         const void *callbackData, PropertyCallbackTypeX
                         *callbackFunction);
```

Purpose

Install a property callback function.

The specified function is called whenever one of the specified properties on one of the specified windows changes in any way. If more than one function is installed for the same property, the functions are called in the reverse order in which they were installed.

If the function is already installed as a callback function, the list of windows and properties that are associated with that function are replaced with those specified by the new installation.

Parameters

Input	**display**	`DisplayPtrX`	A pointer to the display of the X server to which the window belongs.
	windowList	`const WindowX []`	An array of windows on which the properties may exist.
	numberofWindows	unsigned integer	Number of windows in the Window List. This value must be greater than 0.
	propertyList	`const PropertyCallbackTypeX []`	An array of handles to properties for which the callback is called.

(continues)

Parameters (Continued)

	numberofProperties	unsigned integer	Number of properties in the Property List.
	callbackData	generic pointer	Pointer to data to be passed to the callback function. This value is passed to the callback function as the **userData** parameter.
	callbackFunction	`PropertyCallbackTypeX *`	Pointer to the function to be called when the properties change.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

Table 9-4. Status Values for InstallXPropertyCallback

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to one or more parameters. The number of windows argument is 0.
`InvalidDisplayXErr`	-2	The **display** argument is not a valid display. This argument must either be the predefined value `CVIXDisplay` or be the value returned by `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 returned by `CreateXProperty`.
`InsuffMemXErr`	-19	There is insufficient memory to perform the operation.
`BrokenConnectionXErr`	-21	The connection to the X server was broken. This occurs if the remote server terminated.

Parameter Discussion

display must either be the predefined value `CVIXDisplay` or be the value returned by `ConnectToXDisplay`. Use `CVIXDisplay` if the window is on the same display used by LabWindows/CVI.

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 used by LabWindows/CVI. 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 have been obtained with `CreateXProperty`.

To specify a single property, named *prop*, pass the expression `&`*prop* for this parameter and pass `1` for the **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

This function stores the supplied property item with the specified property on the window. Any existing property value is replaced by this value.

To store multiple property items, use the function `PutXWindowPropertyValue`.

Parameters

Input	**display**	`DisplayPtrX`	A pointer to the display of the X server to which the window belongs.
	window	`WindowX`	The window on which the property item is to be stored.
	property	`PropertyHandleX`	Handle of the property to be stored. This value must have been obtained with `CreateXProperty`.
	propertyItem	generic pointer	Property item to be stored 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 the function `GetXPropertySize`.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

NoXErr	0	The function was successful.
InvalidParamXErr	-1	NULL was passed to one or more parameters.
InvalidDisplayXErr	-2	The **display** argument is not a valid display. This argument must either be the predefined value CVIXDisplay or be the value returned by ConnectToXDisplay.
InvalidWindowXErr	-3	The **window** argument is not a valid window.
InvalidPropertyXErr	-4	The **property** argument is not a valid property handle. This argument must be the value returned by CreateXProperty.
InsuffMemXErr	-19	There is insufficient memory to perform the operation.
GeneralXErr	-20	An Xlib function failed for some unknown reason.
BrokenConnectionXErr	-21	The connection to the X server was broken. This occurs if the remote server terminated.

Parameter Discussion

display must either be the predefined value CVIXDisplay or be the value returned by ConnectToXDisplay. Use CVIXDisplay if the window is on the same display used by LabWindows/CVI.

For the **window** parameter, use CVIXRootWindow to access the default root window of the display used by LabWindows/CVI. 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 the XChangeProperty function.

PutXWindowPropertyValue

```
PropLibXErrType status = PutXWindowPropertyValue (DisplayPtrX display,
                         WindowX window, PropertyHandleX property,
                         unsigned int numberofItems, int mode,
                         void *propertyValue);
```

Purpose

This function stores the supplied value with the property on the window.

To store a single property item, you can use the function `PutXWindowPropertyItem`.

Parameters

Input	**display**	DisplayPtrX	A pointer to the display of the X server to which the window belongs.
	window	WindowX	The window on which the property value is to be stored.
	property	PropertyHandleX	Handle of the property to be stored. This value must have been obtained with `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 be stored 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 be written.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidParamXErr`	-1	NULL was passed to one or more parameters. **mode** is not `ReplaceXPropMode`, `PrependXPropMode` or `AppendXPropMode`.
`InvalidDisplayXErr`	-2	The **display** argument is not a valid display. This argument must either be the predefined value CVIXDisplay or be the value returned by `ConnectToXDisplay`.
`InvalidWindowXErr`	-3	The **window** argument is not a valid window.
`InvalidPropertyXErr`	-4	The **property** argument is not a valid property handle. This argument must be the value returned by `CreateXProperty`.
`TypeMismatchXErr`	-12	The actual X type of the property value on the window does not match the type specified for **property**. This can only occur if you set **mode** to append or prepend.
`UnitMismatchXErr`	-13	The actual X format of the property value on the window does not match the unit specified for **property**. This can only occur if you set **mode** to append or prepend.
`OverflowXErr`	-16	Arithmetic overflow occurred with calculations involving the property item sizes and the number of items specified.
`InsuffMemXErr`	-19	There is insufficient memory to perform the operation.
`GeneralXErr`	-20	An Xlib function failed for some unknown reason.
`BrokenConnectionXErr`	-21	The connection to the X server was broken. This occurs if the remote server terminated.

Parameter Discussion

display must either be the predefined value `CVIXDisplay` or be the value returned by `ConnectToXDisplay`. Use `CVIXDisplay` if the window is on the same display used by LabWindows/CVI.

For the **window** parameter, use `CVIXRootWindow` to access the default root window of the display used by LabWindows/CVI. Use `CVIXHiddenWindow` to access the hidden window associated with your application.

The following values are valid for the **mode** parameter:

`ReplaceXPropMode`—Replace the existing property value with the new value.

`PrependXPropMode`—Add the new property value to the beginning of the existing value.

`AppendXPropMode`—Add 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 the `XChangeProperty` function.

RemoveXWindowProperty

`PropLibXErrType` **status = RemoveXWindowProperty** (`DisplayPtrX` **display**,
`WindowX` **window**,
`PropertyHandleX` **property**);

Purpose

Remove the property from a window.

This function deletes the property value and removes the property from the window.

Parameters

Input	**display**	`DisplayPtrX`	A pointer to the display of the X server to which the window belongs.
	window	`WindowX`	The window from which the property is to be removed.
	property	`PropertyHandleX`	Handle of the property to be removed. This value must have been obtained with `CreateXProperty`.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

NoXErr	0	The function was successful.
InvalidParamXErr	-1	NULL was passed to one or more parameters.
InvalidDisplayXErr	-2	The **display** argument is not a valid display. This argument must either be the predefined value CVIXDisplay or be the value returned by ConnectToXDisplay.
InvalidWindowXErr	-3	The **window** argument is not a valid window.
InvalidPropertyXErr	-4	The **property** argument is not a valid property handle. This argument must be the value returned by CreateXProperty.
InsuffMemXErr	-19	There is insufficient memory to perform the operation.
BrokenConnectionXErr	-21	The connection to the X server was broken. This occurs if the remote server terminated.

Parameter Discussion

display must either be the predefined value CVIXDisplay or be the value returned by ConnectToXDisplay. Use CVIXDisplay if the window is on the same display used by LabWindows/CVI.

For the **window** parameter, use CVIXRootWindow to access the default root window of the display used by LabWindows/CVI. 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 the XDeleteProperty function.

UninstallXPropertyCallback

`PropLibXErrType` **status = UninstallXPropertyCallback**
(`PropertyCallbackTypeX` ***callbackFunction**);

Purpose

Uninstall a property callback function.

After a callback function is uninstalled, it is no longer called when properties change. All property callback functions are automatically uninstalled 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`.

Parameters

Input	**callbackFunction**	`PropertyCallbackTypeX*`	The function that was installed with `InstallXPropertyCallback`.

Return Values

The return value indicates the success or failure status of the function call. A negative value indicates an error. The following table shows **status** values.

`NoXErr`	0	The function was successful.
`InvalidCallbackXErr`	-17	The function specified is not installed as a callback.

Chapter 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 Library 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 advantages and limitations listed here 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. The Easy I/O for DAQ library 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 sample programs for the Easy I/O for DAQ library are located in the `cvi\samples\easyio` directory. These sample programs are discussed in the EASYIO section of `cvi\samples.doc`.

Note: ***It is recommended that you 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 that you can easily perform a scan in one step.

In the Data Acquisition Library you may have to use `Lab_ISCAN_Op`, or `SCAN_Op`, or `MDAQ_Start` depending on which DAQ device you are using. Also, if you are using SCXI, there are a number of SCXI specific functions that must be called prior to actually acquiring 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 SCXI module.

Limitations of Using the Easy I/O for DAQ Library

The Easy I/O for DAQ Library currently only works with Analog I/O, Counter/Timers, and simple Digital I/O.

The Easy I/O for DAQ 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 a tree structure according to the types of operations performed. The Easy I/O for DAQ Library function tree is in Table 10-1.

The first- and second-level bold headings in the function tree are names of the function classes. Function classes are groups of related function panels. The third-level headings in plain text are the names of individual function panels. Each Easy I/O for DAQ function panel generates a function call. The actual function names are in bold italics in columns to the right.

Table 10-1. Easy I/O for DAQ Function Tree

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 Check Waveform(s)	***AOCheckWaveforms***
AO Clear Waveform(s)	***AOClearWaveforms***

(continues)

Table 10-1. Easy I/O for DAQ Function Tree (Continued)

Digital Input/Output	
Read From Digital Line	***ReadFromDigitalLine***
Read From Digital Port	***ReadFromDigitalPort***
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***

- The **Analog Input** function class contains all of the functions that perform A/D conversions.
- The **Asynchronous Acquisition** function class contains all of the functions that perform asynchronous (background) A/D conversions.
- The **Analog Output** function class contains all of the functions that perform D/A conversions.
- The **Digital Input/Output** function class contains all of the functions that perform digital input and output operations.
- The **Counter/Timer** function class contains all of 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 writing programs using the Easy I/O for DAQ Library.

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. You can use multiple DAQ devices in one application; to do so, simply 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 containing the analog input channels that are to be sampled.

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 syntax for the Channel String is as follows:

- **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, therefore, 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 must also 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, PC-LPM-16:**
  These devices can only sample input channels in descending order, and you must end with channel 0 (`"3:0"`). If you are using a Lab-PC+ or 1200 product in differential mode, you must use even-numbered channels (`"6,4,2,0"`).

- **If you are using a DAQPad-MIO-16XE-50**:
  You can read the value of the cold junction compensation temperature sensor using the following string as the channel:

  ```
  "cjtemp"
  ```

- **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 either by specifying the corresponding onboard channels or by using the SCXI channel syntax described below. If you operate the modules in multiplexed mode, you must use the SCXI channel syntax.

The SCXI channel syntax is as follows:

- `"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. SCXI channel ranges 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, when a high/low limit command is encountered, those limits are assigned to the following channels until the next high/low limit command is encountered. The High Limit and Low Limit parameters to AISampleChannels are the initial high/low limits. These parameters can be thought of as the left-most high/low limit command.

The following Channel String,

```
"cmd interChannelRate 1000.0; 0:3"
```

specifies that channels 0 through 3 should be sampled at 1000.0 Hz, in other words, there should be 1/1000.0 = 1ms 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 syntax for the command string can be described using the following guide:

- items enclosed in [] are optional
- <number> is an integer or real number
- <LF> is a line-feed character
- ; | <LF> means you may use either ; or <LF> to separate command strings from channel strings
- ! may be used as an optional command separator
- spaces are optional

The syntax for the initial command string that appears before any channels are specified is:

```
"cmd [interChannelRate <number>[!]] [hi <number> [!]low <number>[!]];|<LF>"
```

The syntax for command strings that appear after any channels are specified is:

```
";|<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 containing the analog output channels that are to be driven.

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 and also print. `daqhwov.pdf` is part of a set of `.pdf` files that come with every DAQ device sold by National Instruments.

The syntax for the Channel String is as follows:

- **If you are using a DAQ device without SCXI,** list the channels to be driven, 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.

The SCXI channel syntax is as follows:

```
"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. SCXI channel ranges 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 used 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 through 10
EISA-A2000	2

Easy I/O for DAQ Function Reference

This section describes each function in the Easy I/O for DAQ Library. The function descriptions are arranged alphabetically.

AIAcquireTriggeredWaveforms

```
short error = AIAcquireTriggeredWaveforms (short device, char channelString[],
                                           long numberOfScans,
                                           double scansPerSecond,
                                           double highLimitVolts,
                                           double lowLimitVolts,
                                           double *actualScanRate,
                                           unsigned short triggerType,
                                           unsigned short edgeSlope,
                                           double triggerLevelV,
                                           char triggerSource[],
                                           long pretriggerScans,
                                           double timeLimitsec,
                                           short fillMode, double waveforms[]);
```

Purpose

This function performs a timed acquisition of voltage data from the analog channels specified in the **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 the Trigger Type to sample repetitive waveforms at up to 20 MHz. See the help for the Trigger Type parameter for details.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	Analog input channels that are to be sampled.
	numberOfScans	long integer	Number of scans to be acquired complete. One scan involves sampling every channel in the **channelString** once.
	scansPerSecond	double	Number of scans performed per second. Any particular channel to be scanned at this rate.
	highLimitVolts	double	Maximum voltage to be measured.
	lowLimitVolts	double	Minimum voltage to be measured.
	triggerType	unsigned short integer	The trigger type.
	edgeSlope	unsigned short integer	The edge/slope condition for triggering.
	triggerLevelV	double	Voltage at which the trigger is to occur.
	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	The maximum length of time in seconds to wait for the data.
	fillMode	short integer	Specifies whether the waveforms array are in `GROUP_BY_CHANNEL` or `GROUP_BY_SCAN` mode.
Output	**actualScanRate**	double	The actual scan rate. The actual scan rate may differ slightly from the scan rate you specified, given the limitations of your particular DAQ device.
	waveforms	double array	Array containing the voltages acquired on the channels specified in the **channelString**.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

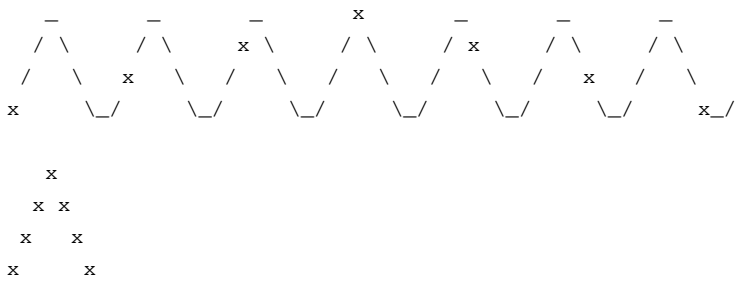
channelString is the analog input channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* subsection of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

triggerType is the trigger type. The trigger types are:

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**, data is retrieved after the analog triggering parameters have been satisfied. Be sure that the Trigger Source is one of the channels listed in the channel string. Hardware triggering is more accurate than software triggering, but it is not available on all boards.
- **If you choose Digital Trigger A**:
  - If **pretriggerScans** is 0, the trigger starts the acquisition. For the MIO-16, connect the digital trigger signal to the START TRIG 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 STOP TRIG 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 START TRIG input starts the acquisition and the STOP TRIG 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**: This is a mode in which the Equivalent Time Sampling technique is used on an E Series DAQ device to achieve an effective acquisition rate of up to 20 MHz.
  - The signal that is being measured must be a periodic waveform.
  - The trigger conditions must be satisfied or this function 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 had been sampled at a high frequency.

For example, if the A/D conversions (represented by `x`'s) on the waveform shown below are placed side-by-side, they represent one cycle of the waveform.

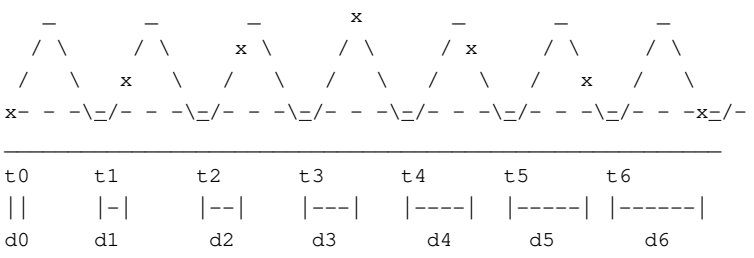


Equivalent Time Sampling is accomplished in this function as follows:

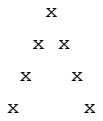
1. Set a hardware analog trigger condition for measuring your waveform using the Edge/Slope, Trigger Level, and Trigger Source parameters of this function.
2. Whenever a hardware analog trigger occurs, the internal ATCOUT signal is strobed.
3. The ATCOUT signal is internally routed to the gate of GPCTR0, which is configured to generate a pulse each time it receives a rising edge at it's gate input.
4. The output of GPCTR0 is internally routed to the data acquisition sample clock to control the A/D conversion rate.
5. The very high effective scan rate is achieved 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.
6. 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.

In the following figure:

`tn` => the nth trigger condition
`dn` => delay between the nth trigger and the nth conversion
`x` => an A/D conversion
`- - -` => the trigger level



When the A/D conversions are placed side-by-side, they represent the original waveform as if it had been sampled at a high frequency.



edgeSlope specifies whether the trigger occurs when the trigger signal voltage is leading (`POSITIVE_SLOPE`) or trailing (`NEGATIVE_SLOPE`).

triggerLevelV the voltage at which the trigger is to occur. **triggerLevelV** is valid only when the Trigger Type is hardware or software analog trigger.

triggerSource specifies which channel is the trigger source. **triggerSource** must be one of the channels listed in the **channelString**. Or if you pass "" or NUL, the first channel in the **channelString** is used as the **triggerSource**. **triggerSource** is valid only when the Trigger Type 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, the function returns a timeout error (`timeOutErr = -10800`).

Other Values:

`-2.0` disables the time limit.

Warning: ***This setting leaves your computer in a suspended state until the trigger condition occurs.***

`-1.0` (default) lets the function calculate the timeout based on the acquisition rate and number of scans requested.

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 Number of Scans is 5, then the possible fill modes are:

```
GROUP_BY_CHANNEL
  A1 A2 A3 A4 A5 B1 B2 B3 B4 B5 C1 C2 C3 C4 C5
   \----------/   \----------/   \----------/
```

or

```
GROUP_BY_SCAN
 A1 B1 C1 A2 B2 C2 A3 B3 C3 A4 B4 C4 A5 B5 C5
  \----/   \----/   \----/   \----/   \----/
```

- If you are to pass the array to a graph, you should acquire the data grouped by channel.
- If you are to pass the array to a strip chart, you should acquire the data grouped by scan.
- You can also acquire the data grouped by scan and later reorder it to be grouped by channel using the `GroupByChannel` function.

waveforms is an array containing the voltages acquired on the channels specified in the **channelString**. The acquired voltages are placed into the array in the order specified by **fillMode**. This array must be declared as large as:

(number of channels) * (**numberOfScans**)

You can determine the number of channels using the `GetNumChannels` function.

AIAcquireWaveforms

`short` **error** = **AIAcquireWaveforms** (`short` **device**, `char` **channelString**`[]`,
`long` **numberOfScans**, `double` **scansPerSecond**,
`double` **highLimitVolts**, `double` **lowLimitVolts**,
`double *`**actualScanRate**, `short` **fillMode**,
`double` **waveforms**`[]`);

Purpose

This function performs a timed acquisition of voltage data from the analog channels specified in the **channelString**.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	Analog input channels that are to be sampled.
	numberOfScans	long integer	Number of scans to be acquired. One scan involves sampling every channel in the **channelString** once.
	scansPerSecond	double	Number of scans performed per second. Any particular channel is scanned at this rate.
	highLimitVolts	double	Maximum voltage to be measured.
	lowLimitVolts	double	Minimum voltage to be measured.
	fillMode	short integer	Specifies one of the following modes for the **waveforms** array: `GROUP_BY_CHANNEL` or `GROUP_BY_SCAN`.
Output	**actualScanRate**	double	The actual scan rate may differ slightly from the scan rate you specified, given the limitations of your particular DAQ device.
	waveforms	double array	Array containing the voltages acquired on the channels specified in the **channelString**.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog input channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* subsection 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 Number of Scans is 5, then the possible fill modes are:

```
GROUP_BY_CHANNEL
   A1 A2 A3 A4 A5 B1 B2 B3 B4 B5 C1 C2 C3 C4 C5
    \----------/   \----------/   \----------/
```

or

```
GROUP_BY_SCAN
  A1 B1 C1 A2 B2 C2 A3 B3 C3 A4 B4 C4 A5 B5 C5
   \----/   \----/   \----/   \----/   \----/
```

- If you are to pass the array to a graph, you should acquire the data grouped by channel.
- If you are to pass the array to a strip chart, you should acquire the data grouped by scan.
- You can also acquire the data grouped by scan and later reorder it to be grouped by channel using the `GroupByChannel` function.

waveforms is an array containing the voltages acquired on the channels specified in the **channelString**. The acquired voltages is placed into the array in the order specified by **fillMode**. This array must be declared as large as:

(number of channels) * (**numberOfScans**)

You can determine number of channels using the function `GetNumChannels`.

AICheckAcquisition

`short` **error** = **AICheckAcquisition** (`unsigned long` **taskID**,
`unsigned long *`**scanBacklog**);

Purpose

This function can be used to determine the backlog of scans that have been acquired into the circular buffer but have not been read using `AIReadAcquisition`.

If `AIReadAcquisition` is called with read mode set to `LATEST_MODE`, **scanBacklog** is reset to zero.

Parameters

Input	**taskID**	unsigned long integer	The task ID that was returned from `AIStartAcquisition`.
Output	**scanBacklog**	unsigned long integer	Returns the backlog of scans that have been acquired into the circular buffer but have not been read using `AIReadAcquisition`.

Return Value

error	short integer	Refer to error codes in Table 10-5.

AIClearAcquisition

`short` **error** = **AIClearAcquisition** (`unsigned long` **taskID**);

Purpose

This function clears the current asynchronous acquisition that was started by `AIStartAcquisition`.

Parameters

Input	**taskID**	unsigned long integer	The task ID that was returned from `AIStartAcquisition`.

Return Value

error	short integer	Refer to error codes in Table 10-5.

AIReadAcquisition

```
short error = AIReadAcquisition (unsigned long taskID, long scanstoRead,
                                 unsigned short readMode,
                                 unsigned long *scanBacklog,
                                 short fillMode, double waveforms[]);
```

Purpose

This function 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	**taskID**	unsigned long integer	The task ID that was returned from AIStartAcquisition.
	scanstoRead	long integer	The number of scans that are read from the internal circular buffer.
	readMode	unsigned short integer	Specifies whether scans are read from the circular buffer in CONSECUTIVE_MODE or LATEST_MODE.
	fillMode	short integer	Specifies one of the following modes for the waveforms array: GROUP_BY_CHANNEL or GROUP_BY_SCAN.
Output	**scanBacklog**	unsigned long integer	Returns the backlog of scans that have been acquired into the circular buffer but have not been read using AIReadAcquisition.
	waveforms	double array	Array containing the voltages acquired on the channels specified in the **channelString**.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

readMode specifies whether scans are read from the circular buffer in CONSECUTIVE_MODE or LATEST_MODE. In CONSECUTIVE_MODE scans are read from the internal circular buffer starting from the last scan that was read. Using this mode, you are guaranteed that you will not lose data unless an error occurs. In LATEST_MODE the most recently acquired n scans are read

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 have not been read using `AIReadAcquisition`. If `AIReadAcquisition` is called in "latest" read mode, the scan backlog is reset to zero. You can also call `AICheckAcquisition` to determine the scan backlog before calling `AIReadAcquisition`.

waveforms is an array containing the voltages acquired on the channels specified in the **channelString**. The acquired voltages are placed into the array in the order specified by **fillMode**. This array must be declared as large as:

(number of channels) * (**scanstoRead**)

You can determine the number of channels by using the function `GetNumChannels`.

AISampleChannel

`short` **error** = **AISampleChannel** (`short` **device**, `char` **singleChannel**`[]`**,**
`double` **highLimitVolts**, `double` **lowLimitVolts**,
`double *`**voltage**);

Purpose

This function acquires a single voltage from a single analog input channel.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	singleChannel	string	The analog input channel that is to be sampled.
	highLimitVolts	double	Maximum voltage to be measured.
	lowLimitVolts	double	Minimum voltage to be measured.
Output	**voltage**	double (passed by reference)	Returns the measured voltage.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

singleChannel is the analog input channel that is to be sampled. See the *Channel String for Analog Input Functions* subsection of the *Easy I/O for DAQ Library Function Overview* section in this chapter for the syntax of this string.

AISampleChannels

```
short error = AISampleChannels (short device, char channelString[],
                                double highLimitVolts, double lowLimitVolts,
                                double voltageArray[]);
```

Purpose

This function performs a single scan on a set of analog input channels.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	Analog input channels that are to be sampled.
	highLimitVolts	double	Maximum voltage to be measured.
	lowLimitVolts	double	Minimum voltage to be measured.
Output	**voltageArray**	double array	Array containing the voltages acquired on the channels specified in the **channelString**.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog input channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* subsection of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

voltageArray is an array containing the voltages acquired on the channels specified in the **channelString**. The acquired voltages are placed into the array in the order specified in the **channelString**. This array must be declared as large as the number of channels specified in the **channelString**. You can use the function `GetNumChannels` to determine the number of channels.

AIStartAcquisition

```
short error = AIStartAcquisition (short device, char channelString[],
                                  int  bufferSize, double scansPerSecond,
                                  double highLimitVolts, double lowLimitVolts,
                                  double *actualScanRate,
                                  unsigned long *taskID);
```

Purpose

This function starts a continuous asynchronous acquisition on the analog input channels specified in the **channelString**. Data is acquired into an internal circular buffer. Use `AIReadAcquisition` to retrieve scans from the internal buffer.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	Analog input channels that are to be sampled.
	bufferSize	integer	The size of the internal circular buffer in scans.
	scansPerSecond	double	Number of scans performed per second. Any particular channel is scanned at this rate.
	highLimitVolts	double	Maximum voltage to be measured.
	lowLimitVolts	double	Minimum voltage to be measured.
Output	**actualScanRate**	double	The actual scan rate may differ slightly from the scan rate you specified, given the limitations of your particular DAQ device.
	taskID	unsigned long integer	An identifier for the asynchronous acquisition.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog input channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* subsection 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 must be passed to

```
AICheckAcquisition
AIReadAcquisition
AIClearAcquisition
```

AOClearWaveforms

`short` **error** = **AOClearWaveforms** (`unsigned long` **taskID**);

Purpose

This function clears the waveforms generated by `AOGenerateWaveforms` when you passed 0 for its **Iterations** parameter.

Parameters

Input	**taskID**	unsigned long integer	The task ID that was returned from `AOGenerateWaveforms`.

Return Value

error	short integer	Refer to error codes in Table 10-5.

AOGenerateWaveforms

```
short error = AOGenerateWaveforms (short device, char channelString[],
                                   double updatesPerSecond,
                                   int updatesPerChannel, int iterations,
                                   double waveforms[],
                                   unsigned long *taskID);
```

Purpose

This function generates a timed waveform of voltage data on the analog output channels specified in the **channelString**.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	The analog output channels to which the voltages are applied.
	updatesPerSecond	double	The number of updates that are performed per second. Any particular channel is updated at this rate.
	updatesPerChannel	integer	The number of D/A conversions that compose a waveform for a particular channel.
	iterations	integer	The number of waveform iterations that are performed before the operation is complete; 0 = continuous.
Output	**waveforms**	double array	The voltages to be applied to the channels specified in the **channelString.**
	taskID	unsigned long integer	Returns an identifier for the waveform generation. If you pass 0 as the **iterations** parameter you need to pass the **taskID** to `AOClearWaveforms` to clear the waveform generation.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog output channels to which the voltages are applied. Refer to the *Channel String for Analog Output Functions* subsection 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, then each waveform is composed of 10 elements from the **waveforms** array.

iterations is the number of waveform iterations that are performed before the operation is complete. If you pass 0, the waveform(s) are generated continuously and you need to call `AOClearWaveforms` to clear waveform generation.

waveforms is the array containing the voltages to be applied to the channels specified in the **channelString**. The voltages are applied to the analog output channels in the order specified in the **channelString**. For example, if the **channelString** is

```
"0:3,5",
```

the array should contain 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

This function applies a specified voltage to a single analog output channel.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	singleChannel	string	The analog output channel to which the voltage are applied.
	voltage	double	The voltage that is applied to the analog output channel.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

singleChannel is the analog output channel to which the voltage are applied. Refer to the *Channel String for Analog Output Functions* subsection 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

This function applies specified voltages to the analog output channel specified in the **channelString**.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	The analog output channels to which the voltages are applied.
	voltageArray	double array	The voltages that are applied to the specified analog output channels.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog output channels to which the voltages are applied. Refer to the *Channel String for Analog Output Functions* subsection 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 specified analog output channels. This array should contain the voltages to be applied to the analog output channels in the order that is specified in the **channelString**. For example, if the **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 TTL pulse train on its OUT pin.

The signal is created by repeatedly decrementing the counter twice, first for the delay to the pulse (phase 1), then for the pulse itself (phase 2). The function selects the highest resolution timebase to achieve the desired characteristics.

You can also call the `CounterStart` function to gate or trigger the operation with a signal on the counter's GATE pin.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	counter	string	The counter to be used for the counting operation.
	frequency	double	The desired repetition rate of the continuous pulse train.
	dutyCycle	double	The desired ratio of the duration of the pulse phase (phase 2) to the period (phase 1 + phase 2).
	gateMode	unsigned short integer	Specifies how the signal on the counter's GATE pin is used.
	pulsePolarity	unsigned short integer	The polarity of phase 2 of each cycle.
Output	**actualFrequency**	double	The achieved frequency based on the resolution and range of your hardware.
	actualDutyCycle	double	The achieved duty cycle based on the resolution and range of your hardware.
	taskID	unsigned long integer	The reference number assigned to this operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 10-2, which is found in the *Valid Counters for the Counter/Timer Functions* subsection 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, the function computes the closest achievable duty cycle using a minimum pulse phase (phase 2) of three timebase cycles.
- If **dutyCycle** = 1.0, the function computes the 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 may not be possible.

gateMode specifies how the signal on the counter's GATE pin is used. The options are:

- `UNGATED_SOFTWARE_START`—ignore the gate signal and start when `CounterStart` is called.
- `COUNT_WHILE_GATE_HIGH`—count while the gate signal is TTL high after `CounterStart` is called.
- `COUNT_WHILE_GATE_LOW`—count while the gate signal is TTL low after `CounterStart` is called.
- `START_COUNTING_ON_RISING_EDGE`—start counting on the rising edge of the TTL gate signal after `CounterStart` is called.
- `START_COUNTING_ON_FALLING_EDGE`—start counting on the falling edge of the TTL gate signal after `CounterStart` is called.

pulsePolarity is the polarity of phase 2 of each cycle. The options are:

- `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 specified counter's SOURCE pin or the number of cycles of a specified internal timebase signal.

When you use this function with the internal timebase and in conjunction with `CounterStart` and `CounterRead` your program can make more precise timing measurements than with the `Timer` function.

You can also call the `CounterStart` function to gate or trigger the operation with a signal on the counter's GATE pin.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	counter	string	The counter to be used for the counting operation.
	counterSize	unsigned short integer	Determines the size of the counter used to perform the operation.
	sourceTimebase	double	`USE_COUNTER_SOURCE`: count TTL edges at **counter's** SOURCE pin; or supply a valid internal timebase frequency to count the TTL edges of an internal clock.
	countLimitAction	unsigned short integer	The action to take when the counter reaches terminal count.
	sourceEdge	short integer	The edge of the counter source or timebase signal on which it increments.
	gateMode	unsigned short integer	Specifies how the signal on the counter's GATE pin is used.
Output	**taskID**	unsigned long integer	The reference number assigned for the counter reserved for this operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 10-2, which is found in the *Valid Counters for the Counter/Timer Functions* subsection of the *Easy I/O for DAQ Library Function Overview* section of this chapter.

counterSize determines the size of the counter used 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 10-3.

Table 10-3. Definition of Am 9513: 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 **counter**'s SOURCE pin, or pass a valid internal timebase frequency to count the TTL edges of an internal clock.

Valid internal timebase frequencies are:

1000000 (Am9513)
100000 (Am9513)
10000 (Am9513)
1000 (Am9513)
100 (Am9513)
20000000 (DAQ-STC)
100000 (DAQ-STC)

countLimitAction is the action to take when the counter reaches terminal count. The parameter 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 this parameter accepts the following attributes:

- `COUNT_ON_RISING_EDGE`
- `COUNT_ON_FALLING_EDGE`

gateMode specifies how the signal on the counter's GATE pin is used. The options are:

- `UNGATED_SOFTWARE_START`—ignore the gate signal and start when `CounterStart` is called.
- `COUNT_WHILE_GATE_HIGH`—count while the gate signal is TTL high after `CounterStart` is called.
- `COUNT_WHILE_GATE_LOW`—count while the gate signal is TTL low after `CounterStart` is called.
- `START_COUNTING_ON_RISING_EDGE`—start counting on the rising edge of the TTL gate signal after `CounterStart` is called.
- `START_COUNTING_ON_FALLING_EDGE`—start counting on the falling edge of the TTL gate signal after `CounterStart` is called.

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 specified counter's SOURCE pin by counting rising edges of the signal during a specified period of time. In addition to this connection, you must also wire the counter's GATE pin to the OUT pin of counter-1. For a specified Counter, Counter-1 and Counter+1 are defined as shown in Table 10-4.

Table 10-4. Adjacent Counters

Am9513		
counter-1	counter	counter+1
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		
counter-1	counter	counter+1
1	0	1
0	1	0

This function is useful for relatively high frequency signals when many cycles of the signal occur during the timing period. Use the `PulseWidthOrPeriodMeasConfig` function for relatively low frequency signals. Keep in mind that

period = 1/**frequency**

This function configures the specified counter and counter+1 (optional) as event counters to count rising edges of the signal on counter's 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 then reads the gate counter until its output state is low. Next the function reads the event counter and computes the signal frequency (number of events/actual gate pulse width) and stops the counters. You can optionally gate or trigger the operation with a signal on counter-1's GATE pin.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	counter	string	The counter to be used for the counting operation.
	counterSize	unsigned short integer	Determines the size of the counter used to perform the operation: `ONE_COUNTER` or `TWO_COUNTERS`.
	gateWidthSampleTimeinSec	double	The desired length of the pulse used to gate the signal. The lower the signal frequency, the longer the Gate Width must be.
	maxDelayBeforeGateSec	double	The maximum expected delay between the time the function is called and the start of the gating pulse. If the gate signal does not start in this time, a timeout occurs.
	counterMinus1GateMode	unsigned short integer	The gate mode for **counter**-1.
Output	**actualGateWidthSec**	double	The length in seconds of the gating pulse that is used.
	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 may 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 may occur at the same time. A timeout does not produce an error.
	frequency	double	The frequency of the signal. It is computed as the (number of rising edges) / (**actualGateWidthSec**).

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 10-2, which is found in the *Valid Counters for the Counter/Timer Functions* subsection of the *Easy I/O for DAQ Library Function Overview* section of this chapter.

counterSize determines the size of the counter used 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 10-3 in the function description for `CounterEventOrTimeConfig`.

counterMinus1GateMode is the gate mode for **counter**-1. The possible values are:

- `UNGATED_SOFTWARE_START`
- `COUNT_WHILE_GATE_HIGH`
- `COUNT_WHILE_GATE_LOW`
- `START_COUNTING_ON_RISING_EDGE`

counter-1 is used 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 10-4.

CounterRead

`short` **error** = **CounterRead** (`unsigned long` **taskID**, `short *`**overflow**, `long *`**count**);

Purpose

Reads the counter identified by **taskID**.

Parameters

Input	**taskID**	unsigned long integer	The reference number assigned to the counting operation by one of the counter configuration functions.
Output	**overflow**	short integer	1 = counter rolled past terminal count; 0 = counter did not roll past terminal count.
	count	long integer	The value of the counter at the time it is read.

Return Value

error	short integer	Refer to error codes in Table 10-5.

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 identified by **taskID**.

Parameters

Input	**taskID**	unsigned long integer	The reference number assigned to the counting operation by one of the counter configuration functions.

Return Value

error	short integer	Refer to error codes in Table 10-5.

CounterStop

`short` **error** = **CounterStop** (`unsigned long` **taskID**);

Purpose

Stops a count operation immediately.

Parameters

Input	**taskID**	unsigned long integer	The reference number assigned to the counting operation by one of the counter configuration functions.

Return Value

error	short integer	Refer to error codes in Table 10-5.

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.

The signal is created by 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 also call the `CounterStart` function to gate or trigger the operation with a signal on the counter's GATE pin.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	counter	string	The counter to be used for the counting operation.
	pulseDelay	double	The desired duration of the delay (phase 1) before the pulse.
	pulseWidth	double	The desired duration of the pulse (phase 2) after the delay.
	timebaseSource	unsigned short integer	The signal that causes the counter to count.
	gateMode	unsigned short integer	Specifies how the signal on the counter's GATE pin is used.
	pulsePolarity	unsigned short integer	The polarity of phase 2 of each cycle.
Output	**actualDelay**	double	The achieved delay based on the resolution and range of your hardware.
	actualPulseWidth	double	The achieved pulse width based on the resolution and range of your hardware.
	taskID	unsigned long integer	The reference number assigned to this operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 10-2, which is found in the *Valid Counters for the Counter/Timer Functions* subsection 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. This parameter accepts the following attributes:

- The unit is seconds if **timebaseSource** is `USE_INTERNAL_TIMEBASE` and cycles if **timebaseSource** is `USE_COUNTER_SOURCE`.
- If **pulseDelay** = 0.0 and **timebaseSource** is internal, the function selects a minimum delay of three cycles of the timebase used.

- **pulseWidth** is the desired duration of the pulse (phase 2) after the delay
- The unit is seconds if **timebaseSource** is `USE_INTERNAL_TIMEBASE` and cycles if **timebaseSource** is `USE_COUNTER_SOURCE`.
- If **pulseDelay** = 0.0 and **timebaseSource** is internal, the function selects a minimum delay of three cycles of the timebase used.

timebaseSource is the signal that causes the counter to count. This parameter accepts the following attributes:

- `USE_INTERNAL_TIMEBASE`—An internal timebase is selected based on the pulse delay and width, in units of seconds.
- `USE_COUNTER_SOURCE`—The signal on the counter's SOURCE pin is used and the units of pulse delay and width are cycles of that signal.

gateMode specifies how the signal on the counter's GATE pin is used. This parameter accepts the following attributes:

- `UNGATED_SOFTWARE_START`—ignore the gate signal and start when `CounterStart` is called.
- `COUNT_WHILE_GATE_HIGH`—count while the gate signal is TTL high after `CounterStart` is called.
- `COUNT_WHILE_GATE_LOW`—count while the gate signal is TTL low after `CounterStart` is called.
- `START_COUNTING_ON_RISING_EDGE`—start counting on the rising edge of the TTL gate signal after `CounterStart` is called.
- `START_COUNTING_ON_FALLING_EDGE`—start counting on the falling edge of the TTL gate signal after `CounterStart` is called.
- `RESTART_ON_EACH_RISING_EDGE`—restart counting on each rising edge of the TTL gate signal after `CounterStart` is called.
- `RESTART_ON_EACH_FALLING_EDGE`—restart counting on each falling edge of the TTL gate signal after `CounterStart` is called.

pulsePolarity is the polarity of phase 2 of each cycle. This parameter accepts the following attributes:

- `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

This function configures the specified counter 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 counter's 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.

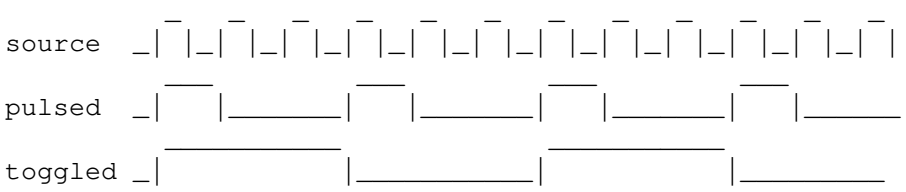
Set the **timebaseDivisor** to the desired value. For a value of N and a pulsed output, an output pulse equal to the period of the source or timebase signal appears on counter's OUT pin once each N cycles of that signal. For a toggled output, the output toggles after each N cycles. The toggled output frequency is thus half that of the pulsed output, in other words,

pulsedFrequency = sourceFrequency/N

and

toggledFrequency = sourceFrequency/(2*N)

thus, if N=3, the OUT pin would generate pulses as follows:



If **gateMode** is not UNGATED_SOFTWARE_START, connect your gate signal to **counter**'s GATE pin.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	counter	string	The counter to be used for the counting operation.
	sourceTimebase	double	`USE_COUNTER_SOURCE`: count TTL edges at **counter's** SOURCE pin; or supply a valid internal timebase frequency to count the TTL edges of an internal clock.
	timebaseDivisor	double	The source frequency divisor.
	gateMode	unsigned short integer	Specifies how the signal on the counter's GATE pin is used.
	outputBehavior	unsigned short integer	The behavior of the output signal when counter reaches terminal count.
	sourceEdge	short integer	The edge of the counter source or timebase signal on which it decrements: `COUNT_ON_RISING_EDGE` or `COUNT_ON_FALLING_EDGE`.
Output	**taskID**	unsigned long integer	The reference number assigned to this operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 10-2, which is found in the *Valid Counters for the Counter/Timer Functions* subsection 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 **counter**'s SOURCE pin, or pass a valid internal timebase frequency to count the TTL edges of an internal clock.

Valid internal timebase frequencies are:

1000000 (Am9513)
100000 (Am9513)
10000 (Am9513)
1000 (Am9513)
100 (Am9513)
20000000 (DAQ-STC)
100000 (DAQ-STC)

timebaseDivisor is the source frequency divisor. For example, if the source signal is 1000 Hz, the **timebaseDivisor** is 10, and the output is pulsed, the frequency of the counter's OUT signal is 100 Hz. If the output is toggled, the frequency is 50 Hz.

gateMode specifies how the signal on the counter's GATE pin is used. This parameter accepts the following attributes:

- `UNGATED_SOFTWARE_START`—ignore the gate signal and start when `CounterStart` is called.
- `COUNT_WHILE_GATE_HIGH`—count while the gate signal is TTL high after `CounterStart` is called.
- `COUNT_WHILE_GATE_LOW`—count while the gate signal is TTL low after `CounterStart` is called.
- `START_COUNTING_ON_RISING_EDGE`—start counting on the rising edge of the TTL gate signal after `CounterStart` is called.
- `START_COUNTING_ON_FALLING_EDGE`—start counting on the falling edge of the TTL gate signal after `CounterStart` is called.

outputBehavior is the behavior of the output signal when counter reaches terminal count. This parameter accepts the following attributes:

- `HIGH_PULSE`—high pulse lasting one cycle of the source or timebase signal.
- `LOW_PULSE`—low pulse lasting one cycle of the source or timebase signal.
- `HIGH_TOGGLE`—high toggle lasting until the next TC.
- `LOW_TOGGLE`—low toggle lasting until the next TC.

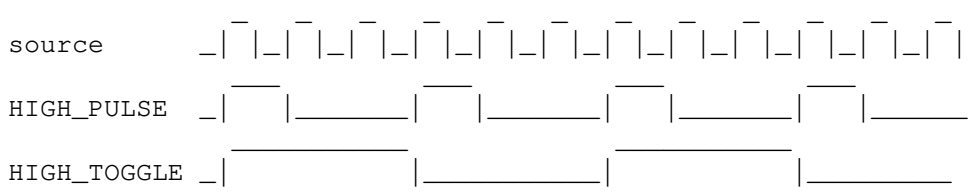
For a Timebase Divisor of `N` and a pulsed output, an output pulse equal to the period of the source or timebase signal appears on counter's OUT pin once each `N` cycles of that signal For a toggled output, the output toggles after each `N` cycles. The toggled output frequency is thus half that of the pulsed output, in other words,

pulsedFrequency = sourceFrequency/ `N`

and

toggledFrequency = sourceFrequency/`(2*N)`

thus, if `N =3`, the OUT pin would generate pulses as follows:



GetAILimitsOfChannel

```
short error = GetAILimitsOfChannel (short device, char channelString[],
                                    char singleChannel[],
                                    double initialHighLimitVolts,
                                    double initialLowLimitVolts,
                                    double *highLimitVolts,
                                    double *lowLimitVolts);
```

Purpose

Returns the high and low limits for a particular channel in the channel string.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	Analog input channels that are to be sampled.
	singleChannel	string	A single channel of the channel string.
	initialHighLimitVolts	double	Specifies the maximum voltage to be measured for all channels in the channel string listed before a command string that specifies a new high limit.
	initialLowLimitVolts	double	The minimum voltage to be measured for all channels in the channel string listed before a command string that specifies a new low limit.
Output	**highLimitVolts**	double	Returns the high limit for the specified channel.
	lowLimitVolts	double	Returns the low limit for the specified channel.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog input channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* subsection 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 could be

`"2"` or `"5"` and so on.

initialHighLimitVolts specifies the maximum voltage that is measured for all channels in the channel string listed before a command string that specifies a new high limit. For 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 that is measured for all channels in the channel string listed before a command string that specifies a new low limit. For 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 the **channelSubString**. For example, if the **channelString** is

```
"1:6"
```

and the **channelSubString** is

```
"1,3,6"
```

the **channelIndices** array would be filled as follows:

channelIndices[0] = 0;

channelIndices[1] = 2;

channelIndices[2] = 5;

This function is useful if you want to verify that a particular channel is part of the **channelString**.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	The analog channel string.
	channelSubString	string	A sub-string of the **channelString**.
	channelType	short integer	Specifies whether the **channelString** is `ANALOG_INPUT` or `ANALOG_OUTPUT`.
Output	**channelIndices**	long integer array	Returns the indices of the channels in the **channelSubString**.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* subsection 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 the **channelString** is

```
"0:3,5"
```

the sub-string could 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 the **channelString** indicated by **index**.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	Analog input channels that are to be sampled.
	index	long integer	The index of a particular channel in the **channelString**.
	channelType	short integer	Specifies whether the **channelString** is `ANALOG_INPUT` or `ANALOG_OUTPUT`.
Output	**channelName**	string	Returns the name of the particular channel in the **channelString** indicated by **index**.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* or *Channel String for Analog Output Functions* subsection 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 the **channelString** indicated by **index**. This string should be declared to have `MAX_CHANNEL_NAME_LENGTH` bytes.

GetDAQErrorString

`char *`**errorString = GetDAQErrorString** (`short` **errorNumber**);

Purpose

This function returns a string containing the description for the numeric error code.

Parameters

Input	**errorNumber**	short integer	The error number that was returned from an Easy I/O for DAQ function.

Return Value

errorString	string	The string containing 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 contained in the **channelString**.

You need to know the number of channels in the **channelString** so that you can interpret (for analog input) or build (for analog output) waveform arrays correctly.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	channelString	string	The analog channel string.
	channelType	short integer	Specifies whether the **channelString** is ANALOG_INPUT or ANALOG_OUTPUT.
Output	**numberOfChannels**	unsigned long integer	Returns the number of channels contained in the **channelString**.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

channelString is the analog channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* or *Channel String for Analog Output Functions* subsection 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

This function can be used to reorder 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 it can be passed to graph plotting functions, analysis functions, and others.

See the description of the **fillMode** parameter of `AIAcquireWaveforms` for an explanation of "grouped by scan" versus "grouped by channel".

Parameters

Input/ Output	**array**	double array	Pass in the “grouped by scan” array and it is grouped by channel in place.
Input	**numberOfScans**	long integer	The number of scans contained in the data array.
	numberOfChannels	unsigned long integer	Specifies the number of channels that were scanned. You can use `GetNumChannels` to determine the number of channels contained in your channel string.

Return Value

error	short integer	Refer to error codes in Table 10-5.

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 (Lab boards, SCXI-1200, DAQPad-1200, PC-LPM-16, DAQCard 700).

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	counter	short integer	The counter to be controlled (valid counters are 0 through 2).
	controlCode	short integer	Determines the counter's operating mode.
	count	unsigned short integer	The period between output pulses.
	binaryorBCD	short integer	`I_BINARY`: The counter operates as a 16-bit binary counter (0 to 65,535); `I_BCD`: The counter operates as a 4-decade BCD counter (0 to 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. Valid when the **controlCode** = 7 (`I_RESET`).
Output	**readValue**	unsigned short integer	Returns the value read from the counter when **controlCode** = 6 (`I_READ`).

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

controlCode determines the counter's operating mode. This parameter accepts the following attributes:

- 0: `I_TOGGLE_ON_TC`—counter's 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 once the counter reaches 0.
- 1: `I_PROGRAMMABLE_ONE_SHOT`—counter's output becomes low on the count following the leading edge of the gate input and becomes high on TC.
- 2: `I_RATE_GENERATOR`—counter's output becomes low for one period of the clock input. The **count** indicates the period between output pulses.
- 3: `I_SQUARE_WAVE_RATE_GENERATOR`—counter's output stays high for one-half of the **count** clock pulses and stays low for the other half.
- 4: `I_SOFTWARE_TRIGGERED_STROBE`—counter's 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—read the counter and return the value in the **readValue** parameter.
- 7: I_RESET—resets the counter and sets its output to **outputState**.

count is the period between output pulses. This parameter accepts the following attributes:

- 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 65,535 in binary counter operation and 10,000 in BCD counter operation.***

PlotLastAIWaveformsPopup

short **error** = **PlotLastAIWaveformsPopup** (short **device**, double **waveformsBuffer**[]);

Purpose

This function plots the last AI waveform that was acquired. It is intended for demonstration purposes.

Data must be grouped by channel before it is passed to this function:

Either use the GROUP_BY_CHANNEL as the *fillMode* parameter when acquiring the data or call GroupByChannel before calling this function.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	waveformsBuffer	double array	Array containing the last AI waveform acquired.

Return Value

error	short integer	Refer to error codes in Table 10-5.

PulseWidthOrPeriodMeasConfig

```
short error = PulseWidthOrPeriodMeasConfig (short device, char counter[],
                                            unsigned short typeOfMeasurement,
                                            double sourceTimebase,
                                            unsigned long *taskID);
```

Purpose

Configures the specified counter to measure the pulse width or period of a TTL signal connected to its GATE pin. The measurement is done by counting the number of cycles of the specified timebase between the appropriate starting and ending events.

Connect the signal you want to measure to the counter's GATE pin.

To measure with an internal timebase, set **sourceTimebase** to the desired frequency.

To measure with an external timebase, connect that signal to **counter**'s SOURCE pin and set the **sourceTimebase** parameter to `USE_COUNTER_SOURCE`.

Call `CounterStart` to start the measurement. Then call `CounterRead` to read the value. A valid *count* value is greater than 3 without overflow.

Parameters

Input	**device**	short integer	Assigned by configuration utility.
	counter	string	The counter to be used for the counting operation.
	typeOfMeasurement	unsigned short integer	Identifies the type of pulse width or period measurement to make.
	sourceTimebase	double	`USE_COUNTER_SOURCE`: count TTL edges at **counter's** SOURCE pin; or supply a valid internal timebase frequency to count the TTL edges of an internal clock.
Output	**taskID**	unsigned long integer	The reference number assigned for the counter reserved for this operation. You pass **taskID** to `CounterStart`, `CounterRead`, and `CounterStop`.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

typeOfMeasurement identifies the type of pulse width or period measurement to make. This parameter 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 **counter's** SOURCE pin, or pass a valid internal timebase frequency to count the TTL edges of an internal clock.

Valid internal timebase frequencies are:

1000000	(Am9513)
100000	(Am9513)
10000	(Am9513)
1000	(Am9513)
100	(Am9513)
20000000	(DAQ-STC)
100000	(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	**device**	short integer	Assigned by configuration utility.
	portNumber	string	Specifies the digital port this function configures.
	line	short integer	Specifies the individual bit or line within the port to be used for I/O (zero-based).
	portWidth	short integer	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.
	configure	long integer	1: Configure the digital port before reading; 0: Don't configure the digital port before reading. When this function is called in a loop, it can be optimized by only configuring the digital port on the first iteration.
Output	**lineState**	unsigned long integer	Returns the state of the digital line. 1 = logical high; 0 = logical low.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

portNumber specifies the digital port this function 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

  `"SCx!MDy!0"`

  syntax, 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.

- The **portWidth** 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) should be at least 8.

configure specifies whether to configure the digital port before reading.

- When this function is called in a loop, it can be optimized by only configuring the digital port 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, where all the ports within the same PPI chip get 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	**device**	short integer	Assigned by configuration utility.
	portNumber	string	Specifies the digital port this function configures.
	line	short integer	Specifies the individual bit or line within the port to be used for I/O.
	portWidth	short integer	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.
	configure	long integer	1: Configure the digital port before reading; 0: Don't configure the digital port before reading. When this function is called in a loop, it can be optimized by only configuring the digital port on the first iteration.
Output	**pattern**	unsigned long integer	The data read from the digital port.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

portNumber specifies the digital port this function 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

`"SCx!MDy!0"`

syntax, 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.
- The **portWidth** 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) should be at least 8.

configure specifies whether to configure the digital port before reading.

- When this function is called in a loop, it can be optimized by only configuring the digital port 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, where all the ports within the same PPI chip get 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, these functions do not reconfigure the hardware unless you change the parameters to the functions. Thus, the performance of these functions is improved by only reconfiguring the hardware when necessary.

However, if you run multiple data acquisition programs simultaneously, any non-timed Easy I/O for DAQ functions will not know when the hardware has been reconfigured by another application accessing the same DAQ device, and the functions will run incorrectly.

To get around this problem, you can force these functions to always reconfigure the hardware 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 be accessing the same board while your program is running. In this mode, the Easy I/O for DAQ functions always reconfigure the hardware on each invocation, which means they will not be adversely affected by other applications but they will not be optimized for speed.

You should set the multitasking mode to `MULTITASK_UNAWARE` if you know there will not be another program accessing the same DAQ device while your program is running.

Parameters

Input	**multitaskingMode**	integer	When activated, 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	**device**	short integer	Assigned by configuration utility.
	portNumber	string	Specifies the digital port this function configures.
	line	short integer	Specifies the individual bit or line within the port to be used for I/O.
	portWidth	short integer	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.
	configure	long integer	1: Configure the digital port before writing; 0: Don't configure the digital port before writing. When this function is called in a loop, it can be optimized by only configuring the digital port on the first iteration.
	lineState	unsigned long integer	Specifies the new state of the digital line. 1 = logical high; 0 = logical low.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

portNumber specifies the digital port this function 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

```
"SCx!MDy!0"
```

syntax, 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.

- The **portWidth** 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) should be at least 8.

configure specifies whether to configure the digital port before writing.

- When this function is called in a loop, it can be optimized by only configuring the digital port 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, where all the ports within the same PPI chip get 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	**device**	short integer	Assigned by configuration utility.
	portNumber	string	Specifies the digital port this function configures.
	portWidth	short integer	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.
	configure	long integer	1: Configure the digital port before writing; 0: Don't configure the digital port before writing. When this function is called in a loop, it can be optimized by only configuring the digital port on the first iteration.
	pattern	unsigned long integer	Specifies the new state of the lines in the port.

Return Value

error	short integer	Refer to error codes in Table 10-5.

Parameter Discussion

portNumber specifies the digital port this function 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

```
"SCx!MDy!0"
```

syntax, 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.
- The **portWidth** 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) should be at least 8.

configure specifies whether to configure the digital port before writing.

- When this function is called in a loop, it can be optimized by only configuring the digital port 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, where all the ports within the same PPI chip get reset to logic low, regardless of the data direction. The data directions on other ports, however, are maintained.

Error Conditions

All of 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 10-5. Easy I/O for DAQ Error Codes

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 specified in the string are inconsistent with other values specified 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** The device parameter is invalid.
-10006	**badLineErr** The 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** The group parameter is invalid.
-10009	**badCounterErr** The counter parameter is invalid.
-10010	**badCountErr** The count parameter is too small or too large for the specified counter.
-10011	**badIntervalErr** The interval parameter is too small or too large for the associated counter or I/O channel.
-10012	**badRangeErr** The analog input or analog output voltage range is invalid for the specified channel.
-10013	**badErrorCodeErr** The driver returned an unrecognized or unlisted error code.
-10014	**groupTooLargeErr** The group size is too large for the device.
-10015	**badTimeLimitErr** The time limit parameter is invalid.
-10016	**badReadCountErr** The read count parameter is invalid.
-10017	**badReadModeErr** The read mode parameter is invalid.
-10018	**badReadOffsetErr** The offset is unreachable.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-10019	**badClkFrequencyErr** The frequency parameter is invalid.
-10020	**badTimebaseErr** The timebase parameter is invalid.
-10021	**badLimitsErr** The limits are beyond the range of the device.
-10022	**badWriteCountErr** Your data array contains an incomplete update, or you are trying to write past the end of the internal buffer, or your output operation is continuous and the length of your array is not a multiple of one half of the internal buffer size.
-10023	**badWriteModeErr** The write mode is out of range or is invalid.
-10024	**badWriteOffsetErr** The write offset plus the write mark is greater than the internal buffer size or it must be set to 0.
-10025	**limitsOutOfRangeErr** The voltage limits are out of range for this device in the current configuration. Alternate limits were selected.
-10026	**badInputBufferSpecification** The 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** The cutoff frequency is not valid for this device.
-10080	**badGainErr** The gain parameter is invalid.
-10081	**badPretrigCountErr** The pretrigger sample count is invalid.
-10082	**badPosttrigCountErr** The posttrigger sample count is invalid.
-10083	**badTrigModeErr** The trigger mode is invalid.
-10084	**badTrigCountErr** The trigger count is invalid.
-10085	**badTrigRangeErr** The trigger range or trigger hysteresis window is invalid.
-10086	**badExtRefErr** The external reference value is invalid.
-10087	**badTrigTypeErr** The trigger type parameter is invalid.
-10088	**badTrigLevelErr** The trigger level parameter is invalid.
-10089	**badTotalCountErr** The total count specified is inconsistent with the buffer configuration and pretrigger scan count or with the device type.
-10090	**badRPGErr** The individual range, polarity, and gain settings are valid but the combination specified is invalid for this device.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-10091	**badIterationsErr** The analog output buffer iterations count is invalid. It must be 0 (for indefinite iterations) or 1.
-10100	**badPortWidthErr** The requested digital port width is not a multiple of the hardware port width.
-10240	**noDriverErr** The driver interface could not locate or open the driver.
-10241	**oldDriverErr** The driver is out of date.
-10242	**functionNotFoundErr** The specified function is not located in the driver.
-10243	**configFileErr** The 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** The driver encountered a hardware-initialization error while attempting to configure the specified device.
-10245	**osInitErr** The 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** The driver is unable to communicate with the specified external device.
-10247	**cmosConfigErr** The CMOS configuration memory for the computer is empty or invalid, or the configuration specified does not agree with the current configuration of the computer.
-10248	**dupAddressErr** The base addresses for two or more devices are the same; consequently, the driver is unable to access the specified device.
-10249	**intConfigErr** The interrupt configuration is incorrect given the capabilities of the computer or device.
-10250	**dupIntErr** The interrupt levels for two or more devices are the same.
-10251	**dmaConfigErr** The DMA configuration is incorrect given the capabilities of the computer/DMA controller or device.
-10252	**dupDMAErr** The DMA channels for two or more devices are the same.
-10253	**switchlessBoardErr** NI-DAQ was unable to find one or more switchless boards you have configured using WDAQCONF.
-10254	**DAQCardConfigErr** Cannot configure the DAQCard because: 1) The correct version of card and socket services software is not installed. 2) The card in the PCMCIA socket is not a DAQCard. 3) The base address and/or interrupt level requested are not available according to the card and socket services resource manager. Try different settings or use AutoAssign in the NIDAQ configuration utility.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-10340	**noConnectErr** No RTSI signal/line is connected, or the specified signal and the specified line are not connected.
-10341	**badConnectErr** The RTSI signal/line cannot be connected as specified.
-10342	**multConnectErr** The specified RTSI signal is already being driven by a RTSI line, or the specified RTSI line is already being driven by a RTSI signal.
-10343	**SCXIConfigErr** The specified SCXI configuration parameters are invalid, or the function cannot be executed given the current SCXI configuration.
-10360	**DSPInitErr** The DSP driver was unable to load the kernel for its operating system.
-10370	**badScanListErr** The 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** The specified resource is owned by the user and cannot be accessed or modified by the driver.
-10401	**unknownDeviceErr** The 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** The 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** The specified line is in use.
-10408	**chanBusyErr** The specified channel is in use.
-10409	**groupBusyErr** The specified group is in use.
-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** The 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.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-10414	**reservedPinErr** Selected signal indicates a pin reserved by NI-DAQ. You cannot configure this pin yourself.
-10440	**sysOwnedRsrcErr** The 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** The specified memory is disabled or is unavailable given the current addressing mode.
-10443	**memAlignmentErr** The 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.
-10444	**memFullErr** No more system memory is available on the heap, or no more memory is available on the device.
-10445	**memLockErr** The transfer buffer cannot be locked into physical memory.
-10446	**memPageErr** The transfer buffer contains a page break; system resources may require reprogramming when the page break is encountered.
-10447	**memPageLockErr** The operating environment is unable to grant a page lock.
-10448	**stackMemErr** The driver is unable to continue parsing a string input due to 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** The driver is unable to make the transfer buffer contiguous in virtual memory and therefore cannot lock the buffer into physical memory; thus, you cannot use the buffer for DMA transfers.
-10452	**noIntAvailErr** No interrupt level is available for use.
-10453	**intInUseErr** The 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** The specified DMA channel is already in use by another device.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-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** The DLL could not be called due to an interface error.
-10460	**interfaceInteractionErr** You have attempted to mix LabVIEW 2.2 VIs and LabVIEW 3.0 VIs. You may run an application consisting only of 2.2 VIs, then run the 2.2 Board Reset VI, before you can run any 3.0 VIs. You may run an application consisting of only 3.0 VIs, then run the 3.0 Device Reset VI, before you can run any 2.2 VIs.
-10560	**invalidDSPhandleErr** The 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** The specified resources have already been configured by a setup operation.
-10602	**noWriteErr** No output data has been written into the transfer buffer.
-10603	**groupWriteErr** The 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.
-10605	**endWriteErr** No data was written to the transfer buffer because the final data block has already been loaded.
-10606	**notArmedErr** The specified resource is not armed.
-10607	**armedErr** The 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** The specified line does not support the specified transfer direction.
-10613	**badChanDirErr** The specified channel does not support the specified transfer direction.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-10614	**badGroupDirErr** The specified group does not support the specified transfer direction.
-10615	**masterClkErr** The clock configuration for the clock master is invalid.
-10616	**slaveClkErr** The clock configuration for the clock slave is invalid.
-10617	**noClkSrcErr** No source signal has been assigned to the clock resource.
-10618	**badClkSrcErr** The specified source signal cannot be assigned to the clock resource.
-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** The specified trigger signal cannot be assigned to the trigger resource.
-10622	**preTrigErr** The 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** The delayed trigger mode is not supported or is not available in the current configuration, or no delay source has been assigned.
-10625	**masterTrigErr** The trigger configuration for the trigger master is invalid.
-10626	**slaveTrigErr** The 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** The 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.
-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.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-10684	**badChanInputModeErr** All channels of this device must have the same input range.
-10685	**clkExceedsBrdsMaxConvRate** The 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** The 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.
-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** The 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** Multi-rate 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.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-10780	**sc2040InputModeErr** When you have an SC2040 attached to your device, all analog input channels must be configured for differential input mode.
-10800	**timeOutErr** The operation could not complete within the time limit.
-10801	**calibrationErr** An error occurred during the calibration process.
-10802	**dataNotAvailErr** The requested amount of data has not yet been acquired, or the acquisition has completed and no more data is available to read.
-10803	**transferStoppedErr** The transfer has been stopped to prevent regeneration of output data.
-10804	**earlyStopErr** The transfer stopped prior to reaching the end of the transfer buffer.
-10805	**overRunErr** The 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** The trigger occurred before sufficient pretrigger data was acquired.
-10809	**gateSignalErr** Attempted to start a pulse width measurement with the pulse in the active state.
-10840	**softwareErr** The contents or the location of the driver file was changed between accesses to the driver.
-10841	**firmwareErr** The firmware does not support the specified operation, or the firmware operation could not complete due to a data-integrity problem.
-10842	**hardwareErr** The hardware is not responding to the specified operation, or the response from the hardware is not consistent with the functionality of the hardware.
-10843	**underFlowErr** The update rate exceeds your system's 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 may 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.

(continues)

Table 10-5. Easy I/O for DAQ Error Codes (Continued)

-10848	**noDMACountAvailErr** The driver could not obtain a valid reading from the transfer-count register in the DMA controller.
-10849	**openFileErr** Unable to open a file.
-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: 1) When waveform generation is in progress, you cannot change the interval timebase. 2) When you make several changes in a row, you must wait long enough for each change to take effect before you request further changes.
-10920	**gpctrDataLossErr** One or more data points may have been lost during buffered GPCTR operations due to speed limitations of your system.

Appendix A
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

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 or (800) 327-3077
  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: 1 48 65 15 59
  Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FaxBack Support

FaxBack is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access FaxBack from a touch-tone telephone at the following number: (512) 418-1111.

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.

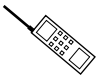
E-Mail Support (currently U.S. only)

You can submit technical support questions to the appropriate applications engineering team through e-mail at the Internet addresses listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

GPIB:	`gpib.support@natinst.com`
DAQ:	`daq.support@natinst.com`
VXI:	`vxi.support@natinst.com`
LabVIEW:	`lv.support@natinst.com`
LabWindows:	`lw.support@natinst.com`
HiQ:	`hiq.support@natinst.com`
VISA:	`visa.support@natinst.com`
Lookout:	`lookout.support@natinst.com`

Fax and Telephone Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	 **Telephone**	 **Fax**
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	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
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (_____)_____ Phone (_____)_____

Computer brand _____ Model _____ Processor _____

Operating system: Windows 3.1, Windows for Workgroups 3.11, Windows NT 3.1, Windows NT 3.5, Windows 95, other (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 will reproduce the problem _____

_____

_____

_____

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. When you complete this form accurately before contacting National Instruments for technical support, our applications engineers can answer your questions more efficiently.

National Instruments Products

Data Acquisition Hardware Revision _____

Interrupt Level of Hardware _____

DMA Channels of Hardware _____

Base I/O Address of Hardware _____

NI-DAQ, LabVIEW, or
LabWindows Version _____

Other Products

Computer Make and Model _____

Microprocessor _____

Clock Frequency _____

Type of Video Board Installed _____

Operating System _____

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: **July 1996**

Part Number: **320682C-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 _____

_____

Fax (_____)_____ Phone (_____)_____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Glossary

Prefix	Meaning	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	Analog input.
A/D	Analog-to-digital.
AC	Alternating current.
ADC A/D converter	An electronic device, often an integrated circuit, that converts an analog voltage to a digital number.
ADC resolution	The resolution of the ADC, which is measured in bits. An ADC with 16 bits has a higher resolution, and thus a higher degree of accuracy, than a 12-bit ADC.
analog trigger	A trigger that occurs at a user-selected point on an incoming analog signal. Triggering can be set to occur at a specific level on either an increasing or a decreasing signal (positive or negative slope). Analog triggering can be implemented either in software or in hardware. When implemented in software, all data is collected, transferred into system memory, and analyzed for the trigger condition. When analog triggering is implemented

	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 prior to the completion or termination of the operation.
automatic serial	A feature in which serial polls are executed automatically by the GPIB polling driver whenever a device asserts the SRQ line.

B

B	Bytes.
background acquisition	Data is acquired by a DAQ system while another program or processing routine is running without apparent interruption.
bipolar	A signal range that includes both positive and negative values (for example, -5 V to +5 V).
block-mode	A high-speed data transfer in which the address of the data is sent followed by a specified number of back-to-back data words.

C

CodeBuilder	The LabWindows/CVI feature that creates code based on a `.uir` file to connect your GUI to the rest of your program. This code is complete and can be compiled and run as soon as it is created.
cold-junction compensation	A method of compensating for inaccuracies in thermocouple circuits.
conversion time	The time required, in an analog input or output system, from the moment a channel is interrogated (such as with a read instruction) to the moment that accurate data is available.
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

D/A	Digital-to-analog.
DAC D/A converter	An electronic device, often an integrated circuit, that converts a digital number into a corresponding analog voltage or current.
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.
DC	Direct current.
device	Device is used to refer to a DAQ device inside your computer or attached directly to your computer via 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 consisting of two terminals, both of which are isolated from computer ground, whose difference is measured.
digital port	See port.
DIO	Digital I/O.

E

external trigger	A voltage pulse from an external source that triggers an event such as A/D conversion.

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

G gain	The factor by which a signal is amplified, sometimes expressed in decibels.
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 that can be attached 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

I/O	Input/output.
ID	Identification.
IEEE	Institute of Electrical and Electronics Engineers.
in.	Inches.
Instrument Library	A LabWindows/CVI library that contains instrument drivers.
interrupt	A computer signal indicating that the CPU should suspend its current task to service a designated activity.

K

KB	Kilobytes of memory.
kS	1,000 samples.
ksamples	1,000 samples.

L

LSB	Least significant bit.

M

manual scaling	Where `SetAxRange` is called to explicitly set the maximum and minimum X and Y values.
MB	Megabytes of memory.
MIO	Multifunction I/O.
ms	Milliseconds.
mux	Multiplexer; a switching device with multiple inputs that sequentially connects each of its inputs to its output, typically at high speeds, in order to measure several signals with a single analog input channel.

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, consisting of four or eight lines of digital input and/or output.
postriggering	The technique used on a DAQ board to acquire a programmed number of samples after trigger conditions are met.

pretriggering	The technique used on a DAQ board to keep a continuous buffer filled with data, so that when the trigger conditions are met, the sample includes the data leading up to the trigger condition.
pts	Points.

R

resolution	The smallest signal increment that can be detected by a measurement system. 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.
RTD	Resistance temperature detector. A metallic probe that measures temperature based upon its coefficient of resistivity.

S

s	Seconds.
S/s	Samples per second; used to express the rate at which a DAQ board samples an analog signal.
Sample-and-Hold (S/H)	A circuit that acquires and stores an analog voltage on a capacitor for a short period of time.
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.
self-calibrating	A property of a DAQ board that has an extremely stable onboard reference and calibrates its own A/D and D/A circuits without manual adjustments by the user.
Single-Ended (SE) Inputs	An analog input that is measured with respect to a common ground.
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.
STC System	Timing Controller.
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.
throughput rate	The data, measured in bytes/s, for a given continuous operation, calculated to include software overhead. Throughput Rate = Transfer Rate - Software Overhead Factor.
transfer rate	The rate, measured in bytes/s, at which data is moved from source to destination after software initialization and set up operations; the maximum rate at which the hardware can operate.

U

unipolar	A signal range that is always positive (for example, 0 to +10 V).

V

V	Volts.
VDC	Volts direct current.

X

Xmodem functions	Allow you to transfer files using a data transfer protocol. The protocol uses a generally accepted technique for performing 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/Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X