LabWindows®/CVI 4.0 Addendum

March 1996 Edition
Part Number 321194A-01

Internet Support

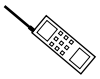
GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabVIEW: lv.support@natinst.com
LabWindows: lw.support@natinst.com
HiQ: hiq.support@natinst.com
VISA: visa.support@natinst.com
FTP Site: ftp.natinst.com
Web Address: www.natinst.com

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

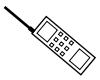
FaxBack Support

(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 48301892, Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 202 2544,
Netherlands 03480 33466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085,
Sweden 08 730 49 70, Switzerland 056 20 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

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

Figures

Tables

About This Manual

The *LabWindows/CVI 4.0 Addendum* describes new features of LabWindows/CVI version 4.0. The information in this manual supplements information in the existing manual set.

Organization of This Manual

The *LabWindows/CVI 4.0 Addendum* is organized as follows.

- Chapter 1, *Updates to the Programmer Reference Manual*, contains updates to the *LabWindows/CVI Programmer Reference Manual*.
- Chapter 2, *Updates to the User Manual*, contains updates to the *LabWindows/CVI User Manual*.
- Chapter 3, *Updates to the User Interface Reference Manual*, contains updates to the *LabWindows/CVI User Interface Reference Manual*.
- Chapter 4, *Updates to the Standard Libraries Reference Manual*, contains updates to the *LabWindows/CVI Standard Libraries Reference Manual*.
- Chapter 5, *General Updates to LabWindows/CVI Manuals*, contains general updates to the LabWindows/CVI documentation.
- Appendix A, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.

The main sections of this manual correspond to specific manuals in the LabWindows/CVI 3.1 manual set. You may want to remove individual sections from the addendum and keep them with the corresponding manual from version 3.1 for convenient, comprehensive reference.

Conventions Used in This Manual

The following conventions are used in this manual:

bold	Bold text denotes menus, menu items, and VI input and output parameters.
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept. Italic text also denotes a variable such as *`filename`* or *N* when it appears in a text passage.

bold italic	Bold italic text denotes a note, caution, or warning.
`monospace`	Monospace font denotes text or characters that you enter using the keyboard. File names, directory names, drive names, sections of code, programming examples, syntax examples, and messages and responses that the computer automatically prints to the screen also appear in this font.
»	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.

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

Related Documentation

The following documents, the manual set for LabWindows/CVI, contain information that you may find helpful as you use this manual.

- *Getting Started with LabWindows/CVI*
- *LabWindows/CVI User Manual*
- *LabWindows/CVI User Interface Reference Manual*
- *LabWindows/CVI Programmer Reference Manual*
- *LabWindows/CVI Instrument Driver Developers Guide*
- *LabWindows/CVI Standard Libraries Reference Manual*
- *LabWindows/CVI Advanced Analysis Reference Manual*
- *C: A Reference Manual*

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

Chapter 1
Updates to the Programmer Reference Manual

Chapter Contents

Compiler/Linker Enhancements for Windows 95 and NT

The compiler/linker capabilities of LabWindows/CVI for Windows 95 and NT are significantly enhanced, compared to LabWindows/CVI for Windows 3.1. A key element is compatibility with four external 32-bit compilers: Microsoft Visual C/C++, Borland C/C++, WATCOM C/C++, and Symantec C/C++. In this manual, the four compilers are referred to as the *compatible external compilers*.

In LabWindows/CVI under Windows 95 and NT, you can now do the following.

- Load 32-bit DLLs, via the standard import library mechanism
- Create 32-bit DLLs and DLL import libraries
- Create library files as well as object files
- Call the LabWindows/CVI libraries from executables or DLLs created in any of the four compatible external compilers
- Create object files, library files, and DLL import libraries that can be used in the compatible external compilers
- Load object files, library files, and DLL import libraries created in any of the compatible external compilers.
- Call Windows SDK functions
- Have a run-time stack of up to 1,000,000 bytes
- Run on a machine without a floating point coprocessor

This chapter discusses these new capabilities.

Loading DLLs in LabWindows/CVI

Chapter 2, *Using Loadable Compiled Modules*, of the *LabWindows/CVI Programmer Reference Manual* discusses the different types of loadable compiled modules and the different ways you can use them. One type of loadable compiled module is a DLL (dynamic link library). Many changes have been made in LabWindows/CVI between Windows 3.1 and Windows 95/NT in how DLLs are handled.

Loading 16-bit DLLs under Windows 3.1

Under Windows 3.1, LabWindows/CVI can load 16-bit DLLs only. Because LabWindows/CVI for Windows 3.1 is a 32-bit program, it cannot use the standard Windows 16-bit DLL import libraries. Instead, LabWindows/CVI needs special "glue code" to link to 16-bit DLLs. In some cases you can specify the DLL file directly in your project, and LabWindows/CVI generates the glue code automatically. In other cases, you modify the glue code source and compile it into an object file, which you substitute for the DLL file in your project. (For more information, see the *DLL Glue Code* section in Chapter 2, *Using Loadable Compiled Modules*, of the *LabWindows/CVI Programmer Reference Manual*.)

If a DLL is being used as an instrument driver or a user library, then either it must be in the same directory as the function panel (`.fp`) file, or there must be a DLL path (`.pth`) file in the same directory. The DLL path file contains the pathname of the DLL. If the pathname contains no directories, the LabWindows/CVI finds the DLL using the standard Windows DLL search algorithm. . (For more information, see the *DLL Search Precedence* section in Chapter 2, *Using Loadable Compiled Modules*, of the *LabWindows/CVI Programmer Reference Manual*.)

Note: ***Starting in LabWindows/CVI version 4.0, if there is no DLL or DLL path file in the same directory as the*** `.fp` ***file, LabWindows/CVI looks for a DLL with the same base name as the*** `.fp` ***file using the standard Windows search algorithm.***

Loading 32-bit DLLs under Windows 95 and NT

Under Windows 95 and NT, LabWindows/CVI can load 32-bit DLLs. Because the environment is 32-bit, special glue code is no longer needed. LabWindows/CVI links to DLLs via the standard 32-bit DLL import libraries that you generate when you create 32-bit DLLs with any of the compilers. Because DLLs are linked in this way, you can no longer specify a DLL file directly in your project. You must specify the DLL import library file instead.

DLLs for Instrument Drivers and User Libraries

Under Windows 95 and NT, a DLL is never directly associated with an instrument driver or user library. Instead, an instrument driver or user library can be associated with a DLL import library.

In general, if the program for an instrument driver or user library is contained in a DLL, there must be a DLL import library in the same directory as the function panel (`.fp`) file. An exception is made to facilitate using VXI Plug & Play instrument driver DLLs. When you install a VXI Plug & Play instrument driver, the DLL import library is not placed in the same directory as the `.fp` file. If a `.fp` file is under the VXI Plug & Play directory, LabWindows/CVI searches for an import library in the VXI Plug & Play import library directory before it looks for a program file in the directory of the `.fp` file.

Using The LoadExternalModule Function

When using the `LoadExternalModule` function to load a DLL at run-time, you must specify the pathname of the DLL import library, not the name of the DLL.

Link Errors when Using DLL Import Libraries

A DLL import library must not contain any references to symbols that are not exported by the DLL. If it does, LabWindows/CVI reports a link error. (If you load the DLL using `LoadExternalModule`, the `GetExternalModuleAddr` function reports an undefined references (-5) error.) You can solve this problem by using LabWindows/CVI to generate an import library. See *Generating an Import Library* later in this section.

DLL Path (.pth) Files No Longer Supported

The DLL import library contains the file name of the DLL. LabWindows/CVI uses the standard Windows DLL search algorithm to find the DLL. Thus, DLL path (`.pth`) files do not work under Windows 95 and NT.

16-Bit DLLs No Longer Supported

LabWindows/CVI for Windows 95 and NT does not load 16-bit DLLs. If you want to do so, you must obtain a 32-to-16-bit thunking DLL and a 32-bit DLL import library.

Generating an Import Library

If you do not have a DLL import library or the one you have contains references not exported by the DLL, you can generate an import library in LabWindows/CVI. You must have an include file that contains the declarations of all of the functions and global variables you want to access

from the DLL. Load the include file into a Source window, and execute the **Generate DLL Import Library** command in the **Options** menu.

Default Unloading/Reloading Policy

Between Windows 3.1 and Windows 95/NT, some fundamental changes were made in the way DLLs being used by multiple processes are handled.

In Windows 3.1, a DLL being used by multiple processes has only one data space. In Windows 95/NT, a separate data space is created for each process that is using the DLL.

In Windows 3.1, a DLL is not notified each time it is loaded or unloaded by a process. It is notified only when the first process loads it and the last process unloads it. In Windows 95/NT, a DLL is notified each time it is loaded or unloaded by a process.

In LabWindows/CVI for Windows 3.1, DLLs are, by default, kept in memory between executions of user programs in the development environment. The purpose is to save the time it takes to reload DLLs on each run. Because Windows 3.1 DLLs cannot rely on being notified that they are being loaded or unloaded, they should be able to operate correctly under such conditions. You can cause DLLs to be reloaded before each run by setting the **Reload DLLs Before Each Run** option in the **Run Options** dialog box.

In LabWindows/CVI for Windows 95 and NT, the default has been changed. DLLs are, by default, unloaded after each execution of a user program in the development environment. This behavior more accurately simulates what happens when you execute a standalone executable, and it is more suitable for Windows 95 and NT DLLs that rely on load/unload notification on each execution of a program. You can change the default behavior by turning off the **Unload DLLs After Each Run** option in the **Run Options** dialog box.

Compatibility with External Compilers

LabWindows/CVI for Windows 95 and NT can be compatible at the object code level with any of the four compatible external compilers (Microsoft Visual C/C++, Borland C/C++, WATCOM C/C++, and Symantec C/C++). Because these compilers are not compatible each other at the object code level, LabWindows/CVI can be compatible with only one external compiler at a time. In this manual, any of the four compilers are referred to *selected compatible compiler*.

Choosing Your Compatible Compiler

When installing LabWindows/CVI, you must choose your compatible compiler. If sometime later you want to change your choice of compatible compiler, you can run the installation program and choose the option to change your compatible compiler.

The selected compatible compiler is indicated in the dialog box that appears when you execute the **Compiler Options** command in the **Options** menu of the Project window.

Object Files, Library Files, and DLL Import Libraries

If you create an object file, library file, or DLL import library in LabWindows/CVI, the file can be used only in the selected compatible compiler or in LabWindows/CVI under the same compatibility choice.

If you load an object file, library file, or DLL import library file in LabWindows/CVI, the file must have been created either in the selected compatible compiler or in LabWindows/CVI under the same compatibility choice. If you have a DLL but you do not have a compatible DLL import library, you can create one in LabWindows/CVI. You must have an include file that contains the declarations of all of the functions and global variables you want to access from the DLL. Load the include file into a Source window, and execute the **Generate DLL Import Library** command in the **Options** menu.

DLLs

In general, a DLL can be used without regard to compiler used to create it. Only the DLL import library must have been created using the correct compiler or compatibility choice. There are some cases, however, in which a DLL created using one compiler cannot be used in an executable or DLL created using another compiler. If you want to create DLLs that can be used in different compilers, you should design the API for your DLL to avoid such problems. The following are the areas in which the DLLs created in external compilers are not fully compatible.

Structure Packing

The compilers differ in their default maximum alignment of elements within structures.

If your DLL API uses structures, you should guarantee compatibility among the different compilers by using the `pack` pragma to specify a specific maximum alignment factor.

You should do this in the DLL include file, before the definitions of the structures. (The particular alignment factor you set does not matter.) After the structure definitions, you should reset the maximum alignment factor back to the default. Example:

```
#pragma pack (4)  /* set maximum alignment to 4 */

typedef struct {
      char a;
      int b;
   } MyStruct1;

typdef struct {
      char a;
```

```
      double b;
   } MyStruct2;

#pragma pack ()  /* reset max alignment to default */
```

Bit Fields

Borland C/C++ uses the smallest number of bytes needed to hold the bit fields specified in a structure. The other compilers always use four-byte elements. You can force compatibility by adding a dummy bit field of the correct size to pad the set of contiguous bit fields to fit exactly into a four-byte element. Example:

```
typedef struct {
    int  a:1;
    int  b:1;
    int  c:1;
    int  dummy:29;      /* pad to 32 bits */
} MyStruct;
```

Returning Floats and Doubles

The compilers return `float` and `double` scalar values using different mechanisms. This is true of all calling conventions, including `__stdcall`. The only solution for this problem is to change your DLL API to use output parameters instead of return values for `double` and `float` scalars.

Returning Structures

For functions not declared with the `__stdcall` calling convention, the compilers return structures using different mechanisms. For functions declared with `__stdcall`, the compilers return structures in the same way, except for 8-byte structures. We recommend that your DLL API use structure output parameters instead of structure return values.

Enum Sizes

By default, WATCOM uses the smallest integer size (1-byte, 2-bytes, or 4-bytes) needed to represent the largest `enum` value. The other compilers always use four bytes. You should force compatibility by using the `-ei` (Force Enums to Type Int) option with the WATCOM compiler.

Long Doubles

In Borland C/C++, `long double` values are ten bytes. In the other compilers, they are eight bytes. (In LabWindows/CVI, they are always eight bytes). You should avoid using `long double` in your DLL API.

Differences with the External Compilers

LabWindows/CVI does not work with all of the non-ANSI extensions provided by each external compiler. Also, in cases where ANSI does not specify the exact implementation, LabWindows/CVI does not always agree with the external compilers. Most of these differences are obscure and rarely encountered. The following are the most important differences.

- `wchart_t` is only one-byte in LabWindows/CVI.
- 64-bit integers do not exist in LabWindows/CVI.
- `long double` values are 10 bytes in Borland C/C++ but 8 bytes in LabWindows/CVI.
- You cannot use structured exception handling in LabWindows/CVI.
- You cannot use the WATCOM C/C++ `cdecl` calling convention in LabWindows/CVI for functions that return `float` or `double` scalar values or structures. (In WATCOM, `cdecl` is *not* the default calling convention.)

External Compiler Versions Supported

The following versions of each external compiler work with LabWindows/CVI for Windows 95 and NT:

- Microsoft Visual C/C++, version 2.2 or higher
- Borland C/C++, version 4.51 or higher
- WATCOM C/C++, version 10.5 or higher
- Symantec C/C++, version 7.2 or higher

Required Preprocessor Definitions

When using an external compiler to compile source code that includes any of the LabWindows/CVI include files, add the following to your Preprocessor Definitions.

```
_NI_mswin32_
```

Creating Executables and DLLs in External Compilers for Use with the LabWindows/CVI Libraries

Under Windows 95 and NT, you can use the LabWindows/CVI libraries in any of the four compatible external compilers. You can create executables and DLLs that call the LabWindows/CVI libraries. All of the libraries are contained in DLLs. (These DLLs are also used by executable files created in LabWindows/CVI.) DLL import libraries and a startup library, all compatible with your external compiler, are in the `cvi\extlib` directory. Never use the `.lib` files in the `cvi\bin` directory.

You must always include the following two libraries in your external compiler project.

```
cvisupp.lib  /* startup library */
cvirt.lib    /* import library to DLL containing: */
             /*     User Interface Library        */
             /*     Formatting and I/O Library    */
             /*     RS-232 Library                */
             /*     DDE Library                   */
             /*     TCP Library                   */
             /*     Utility Library               */
```

You may also add the following static library file from `cvi\extlib` to your external compiler project.

```
analysis.lib /* Analysis or Advanced Analysis Library */
```

You may also add the following DLL import library files from `cvi/extlib` to you external compiler project.

```
gpib.lib     /* GPIB/GPIB 488.2 Library */
dataacq.lib  /* Data Acquisition Library */
easyio.lib   /* Easy I/O for DAQ Library */
visa.lib     /* VISA Transition Library */
nivxi.lib    /* VXI Library */
```

If you are using an instrument driver that makes references to both the GPIB and VXI libraries, you can include both `gpib.lib` and `nivxi.lib` to resolve the references to symbols in those libraries. If you do not have access to one of these files, you can replace it with one of following files:

```
gpibstub.obj /* stub GPIB functions */
vxistub.obj  /* stub VXI functions */
```

Include Files for the ANSI C Library and the LabWindows/CVI Libraries

The `cvirt.lib` import library contains symbols for all of the LabWindows/CVI libraries, except the ANSI C standard library. When you create an executable or DLL in an external compiler, you use the compiler's own ANSI C standard library. Because of this, you must use the external compiler's include files for the ANSI C library when compiling source files. Although the include files for the other LabWindows/CVI libraries are in the `cvi\include` directory, the LabWindows/CVI ANSI C include files are in the `cvi\include\ansi` directory. Thus, you can specify `cvi\include` as an include path in your external compiler and still use the external compiler's version of the ANSI C include files.

Note: ***You need to use the external compiler's ANSI C include files only when compiling a source file that you intend to link using the external compiler. If you intend to link it in LabWindows/CVI, you need to use the LabWindows/CVI ANSI C include files. This holds true regardless of which compiler you use to compile the source file.***

For more information, see the *Setting Up Include Paths for LabWindows/CVI, ANSI C, and SDK Libraries* section later in this chapter.

Standard Input/Output Window

One effect of using the external compiler's ANSI C standard library, is that the `printf` and `scanf` functions do not use the LabWindows/CVI Standard Input/Output window. If you want to use `printf` and `scanf`, you must create a console application (called a character-mode executable by WATCOM).

You can continue to use the LabWindows/CVI Standard Input/Output Window by calling the `FmtOut` and `ScanIn` functions in the Formatting and I/O library.

Multithreading and the LabWindows/CVI Libraries

If you are using multithreading in an external compiler, you need to know which of the LabWindows/CVI libraries are multithreaded safe.

Multithreaded Safe Libraries

The following libraries can be used in more than one thread at a time:

- Analysis and Advanced Analysis
- GPIB (if you are using a native 32-bit driver)

- VXI
- VTL
- RS-232
- Data Acquisition
- Easy I/O for DAQ

Also, each of the compatible external compilers includes a multithreaded safe version of the ANSI C standard library.

Note: ***Although you can use Windows SDK function to create threads in a LabWindows/CVI program, none of LabWindows/CVI libraries are multithreaded safe when called from programs linked in LabWindows/CVI.***

Libraries that are Not Multithreaded Safe

Currently, the following LabWindows/CVI libraries must be used in only one thread at a time:

- User Interface
- Formatting and I/O
- DDE
- TCP
- Utility
- GPIB (if you are using a “Windows 3.1 compatibility” driver)

Resolving Callback References From .UIR Files

When you link your program in LabWindows/CVI, LabWindows/CVI keeps a table of the non-static functions that are in your project. When your program calls `LoadPanel` or `LoadMenuBar`, the LabWindows/CVI User Interface Library uses this table to find the callback functions associated with the objects being loaded from the User Interface Resource (`.uir`) file. This is true whether you are running your program in the LabWindows/CVI development environment or as a standalone executable.

When you link your program in an external compiler, no such table is made available to the User Interface Library. Instead, you must use LabWindows/CVI to generate an object file containing the necessary table. Create a LabWindows/CVI project containing the .uir files used by your program (if you do not already have one). Execute the **External Compiler Support** command in the **Build** menu of the Project window. A dialog box appears. In the **UIR Callbacks Object File** control, enter the pathname of the object file to be generated. When you click on the **Create**

button, the object file is generated with a table containing the names of all of the callback functions referenced in all of the `.uir` files in the currently loaded project. If the project is loaded and you modify and save any of these `.uir` files, the object file is regenerated to reflect the changes. You must include this object file in the external compiler project you use to create the executable. Also, you must call `InitCVIRTE` at the beginning of your `main` or `WinMain` function. See the *`InitCVIRTE` and `CloseCVIRTE`* section later in this chapter.

Linking to Callback Functions Not Exported From a DLL

Normally, the User Interface Library searches for callback functions only in the table of functions in the executable. When you load a panel or menu bar from a DLL, you may want to link to non-static callback functions contained in, but not exported by, the DLL. You can do this by calling the `LoadPanelEx` and `LoadMenuBarEx` functions. When you pass the DLL module handle to `LoadPanelEx` and `LoadMenuBarEx`, the User Interface Library searches the table of callback functions contained in the DLL before searching the table contained in the executable. Refer to Chapter 3 of this document for detailed information on `LoadPanelEx` and `LoadMenuBarEx`.

If you create your DLL in LabWindows/CVI, the table of functions is included in the DLL automatically. If you create your DLL using an external compiler, you must generate an object file containing the necessary table. Create a LabWindows/CVI project containing the .uir files loaded by your DLL (if you do not already have one). Execute the **External Compiler Support** command in the **Build** menu of the Project window. A dialog box appears. In the **UIR Callbacks Object File** control, enter the pathname of the object file to be generated. When you click on the **Create** button, the object file is generated with a table containing the names of all of the callback functions referenced in all of the `.uir` files in the currently loaded project. If the project is loaded and you modify and save any of these `.uir` files, the object file is regenerated to reflect the changes. You must include this object file in the external compiler project you use to create the DLL. Also, you must call `InitCVIRTE` and `CloseCVIRTE` in your `DLLMain` function. See the *`InitCVIRTE` and `CloseCVIRTE`* section later in this chapter.

Resolving References from Modules Loaded at Run-Time

Note: ***This section does not apply if you are using*** `LoadExternalModule` ***to load only DLLs (via DLL import libraries).***

Unlike DLLs, object and static library files can contain unresolved references. If you call `LoadExternalModule` to load an object or static library file at run-time, the Utility Library must resolve those references using function and variable symbols from the executable or from previously loaded run-time modules. A table of symbols must be available in the executable. When you link your program in LabWindows/CVI, a symbol table is automatically included. This is true whether you are running your program in the LabWindows/CVI development environment or as a standalone executable.

When you link your program in an external compiler, no such table is made available to the Utility Library; LabWindows/CVI makes available two object files for this purpose.

- Include `cvi\extlib\refsym.obj` in your external compiler project if your run-time modules reference any symbols in the User Interface, Formatting and I/O, RS-232, DDE, TCP, or Utility Library.
- Include `cvi\extlib\arefsym.obj` your external compiler project if your run-time modules reference any symbols in the ANSI C standard library. (If you need to use this object file and you are using Borland C/C++ to create your executable, you must choose Static Linking for the Standard Libraries. In the IDE, this can be done in the New Target and Target Expert dialog boxes.)

Resolving References to Non-CVI Symbols

If your run-time modules reference any other symbols from your executable, you must use LabWindows/CVI to generate an object file containing a table of those symbols. Create an include file containing complete declarations of all of the symbols your run-time modules reference from the executable. The include file may contain nested `#include` statements and may contain executable symbols that your run-time modules do not reference. If your run-time module references any of the commonly used Windows SDK functions, you can use the `cvi\sdk\include\basicsdk.h` file.

Execute the **External Compiler Support** command in the **Build** menu of the Project window. A dialog box appears. Checkmark the **Using Load External Module** checkbox. The **Other Symbols** checkbox should already be checkmarked. Enter the pathname of the include file in the **Header File** control. Enter the pathname of the object file to be generated in the **Object File** control. Click on the **Create** button to the right of the **Object File** control.

Include the object file in the external compiler project you use to create your executable. Also, you must call `InitCVIRTE` at the beginning of your `main` or `WinMain` function. See the *`InitCVIRTE` and `CloseCVIRTE`* section later in this chapter.

Resolving Run-Time Module References to Symbols Not Exported From a DLL

Normally, the Utility Library `LoadExternalModule` function resolves run-time module references using only symbols in your executable or previously loaded run-time modules. When you load an object or static library file from a DLL, you may want to resolve references from that module using non-static symbols contained in, but not exported by, the DLL. You can do this by calling the `LoadExternalModuleEx` function. When you pass the DLL module handle to `LoadExternalModuleEx`, the Utility Library searches the symbol table contained in the DLL before searching the table contained in the executable. Refer to Chapter 4 of this document for detailed information on `LoadExternalModuleEx`.

If you create your DLL in LabWindows/CVI, the table of symbols is included in the DLL automatically. If you create your DLL using an external compiler, no such table is made available to the Utility Library. You must include in your DLL one or more object files containing the necessary symbol tables. You can do this for a DLL in the same manner as for an executable, which is described previously in this section. You must call `InitCVIRTE` and `CloseCVIRTE` in your `DLLMain` function. See the *`InitCVIRTE` and `CloseCVIRTE`* section later in this chapter.

Run State Change Callbacks Are Not Available in External Compilers

When you use a compiled module in LabWindows/CVI, you can arrange for it to be notified of a change in the execution status (start, stop, suspend, resume). This is done through a callback function, which is always named `__RunStateChangeCallback`. This is described in detail in the section *Special Considerations When Using a Loadable Compiled Module*, in Chapter 2, *Using Loadable Compiled Modules*, of the *LabWindows/CVI Programmer Reference Manual*.

The run state change callback capability is necessitated by the fact that when you run a program in the LabWindows/CVI development environment, it is executed as part of the LabWindows/CVI process. When your program terminates, the operating system does not clean up as it does when a process terminates. LabWindows/CVI performs cleans up as much as it can, but your compiled module may need to do more. Also, if the program is suspended for debugging purposes, your compiled module may need to disable interrupts.

When you run an executable created in an external compiler, it is always executed as a separate process, even if you are debugging it. Thus, the run state change callback facility is not needed and does not work. When linking with an external compiler, having a function called `__RunStateChangeCallback` in more than one object file causes a link error. If you need a run state change callback in a compiled module that you intend to use both in LabWindows/CVI and an external compiler, it is recommended that you put it in a separate source file and create a `.lib` file instead of an `.obj` file.

Calling InitCVIRTE and CloseCVIRTE

If you link an executable (or DLL) in an external compiler, you may need to call the `InitCVIRTE` function at the beginning of your `main` or `WinMain` (or `DLLMain`) function. The call is necessary if you have functions in your executable (or non-exported functions in your DLL) that are needed to resolve callback references from `.uir` files or needed to resolve external references in `.obj` or `.lib` files loaded using `LoadExternalModule`. See the *Resolving Callback References From .UIR Files* and *Resolving References from Modules Loaded at Run-Time* sections earlier in this chapter.

For an executable, the code should be as follows.

```
#include <cvirte.h>
main()   /* or WinMain(…….) */
{
   if (InitCVIRTE(0) == 0)
      {
      return (-1);       /* out of memory */
      }

   /* your other code */
}
```

For a DLL, you also need to call `CloseCVIRTE` in `DLLMain`. The code should be as follows.

```
#include <cvirte.h>

int __stdcall DLLMain (void *hinstDLL, int fdwReason, void *lpvReserved)
{
   if (fdwReason == DLL_PROCESS_ATTACH)
      {
      if (InitCVIRTE (hinstDLL) == 0)
         return 0;     /* out of memory */
      /* your other ATTACH code */
      }
   else if (fdwReason == DLL_PROCESS_DETACH)
      {
      /* your other DETACH code */
      CloseCVIRTE ();
      }

   return 1;
}
```

Note: ***It is harmless, but unnecessary, to call these functions when you link your executable in LabWindows/CVI.***

Creating Object and Library Files in External Compilers for Use in LabWindows/CVI

When you use an external compiler to create an object or library file for use in LabWindows/CVI, you must use the include files in the `cvi\include` and `cvi\sdk\include` directories. Be sure that these directories have priority over the default paths for the compiler's C library and SDK library include files.

When you use an external compiler to create an object or library file for use in LabWindows/CVI, you must choose the compiler options carefully. For all compilers, LabWindows/CVI is designed to work with the default options as much as possible. In some

cases, however, you need to choose options that override the defaults. Additionally, there may be some defaults which you must not override.

Microsoft Visual C/C++

LabWindows/CVI is compatible with all of the defaults.

You should *not* use the following options to override the default settings:

`/J`	(Unsigned Characters)
`/Zp`	(Struct Member Alignment)
`/Ge`	(Stack Probes)
`/Gh`	(Profiling)
`/Gs`	(Stack Probes)

Borland C/C++ command line compiler

LabWindows/CVI is compatible with all of the defaults.

You should *not* use the following options to override the default settings:

`-a`	(Data Alignment)
`-K`	(Unsigned Characters)
`-u-`	(Turn Off Generation of Underscores)
`-N`	(Test Stack Overflow)
`-p`	(Pascal Calling Convention)
`-pr`	(Register Calling Convention)

Correct Pentium FDIV Flaw

WATCOM C/C++

You must use the following options to override the default settings:

`-ei`	(Force Enums to Type Int)
`-zw`	(Compile for Windows)
`-4s`	(80486 Stack-Based Calling)
`-s`	(Disable Stack Depth Checking)
`-j`	(Change Char Default to Signed)
`-fpi87`	(Generate In-Line 80x87 Code)

You should *not* use the following option to override the default settings:

`-Zp`	(Structure Alignment)

Symantec C/C++

You must use the following options to override the default settings:

`-mn`	(Windows 95/NT Memory Model)
`-f`	(Generate In-Line 80x87 Code)

You should *not* use the following options to override the default settings:

`-a`	(Struct Alignment)
`-P`	(Use Pascal Calling Convention)
`-s`	(Check Stack Overflow)

Note: ***Certain specialized options may generate symbol references that cause link errors in LabWindows/CVI. If you encounter a link error on a symbol in an externally compiled module and you do not recognize the symbol, try changing your external compiler options.***

Creating Executables in LabWindows/CVI

You can create true 32-bit Windows executables in LabWindows/CVI for Windows 95 and NT. In LabWindows/CVI for Windows 3.1, standalone programs are run using a special executable file that contains the LabWindows/CVI run-time libraries. If you run more than one program at a time, extra copies of this special executable are loaded into memory. Under Windows 95 and NT, the LabWindows/CVI run-time libraries come in DLL form. The same DLLs are used by standalone executables created in LabWindows/CVI and executables created in external compilers. If more than one program is run at a time, only one copy of the DLL is loaded.

To create a standalone executable, you must first select **Standalone Executable** from the submenu attached to the **Target** command in the **Build** menu of the Project window. When **Standalone Executable** is checkmarked, the **Create Standalone Executable** command appears below the Target command in the **Build** menu. The **Create Standalone Executable** command in Windows 95 and NT is the same as in Windows 3.1, except that you can specify version information to be included in the executable in the form of a standard Windows version resource.

Creating DLLs in LabWindows/CVI

In LabWindows/CVI for Windows 95 and NT, you can create 32-bit DLLs. Along with each DLL, LabWindows/CVI creates a DLL import library for your compatible compiler. You can choose to create DLL import libraries compatible with all four compatible external compilers.

You need a separate project for each DLL you want to create. You must select **Dynamic Link Library** from the submenu attached to the **Target** command in the **Build** menu of the Project window. When **Dynamic Link Library** is checkmarked, the **Create Dynamic Link Library** command appears below the **Target** command in the **Build** menu. Refer to Chapter 2, *Updates to the User Manual*, of this document for detailed information on the **Create Dynamic Link Library** command.

There is no provision for debugging DLLs created in LabWindows/CVI.

Customizing an Import Library

If you need to perform some special processing in your DLL import library, you can customize it. Instead of generating a `.lib` file, you can generate a `.c` file containing source code. If you do this, however, you can export only functions from the DLL, not variables.

To customize an import library, you must have an include file that contains the declarations of all of the functions you want to access from the DLL. Load the include file into a Source window, and execute the **Generate DLL Import Source** command in the **Options** menu.

After you have generated the glue source, you can modify it, including making calls to functions in other source files. Create a new project containing the glue source file and any other files it references. Select **Static Library** from the submenu attached to the **Target** command in the **Build** menu of the Project window. Execute the **Create Static Library** command.

Note: ***This glue source code does not operate in the same way as a normal DLL import library. When you link a normal DLL import library into an executable, the operating system attempts to load the DLL as soon as the program starts. The glue source generated by LabWindows/CVI is written so that the DLL is not loaded until the first function call into it is made.***

Preparing Source Code for Use in a DLL

When you create a DLL, you must address the following issues because they can affect your source code and include file.

- The calling convention you use for the exported functions.
- How you specify which DLL functions and variables are to be exported.
- Marking symbols that are imported in the DLL include file you distribute.

This section discusses how you can address these issues when you create your DLL in LabWindows/CVI. If you create your DLL in an external compiler, the approach is very similar. The external compilers, however, do not agree in all aspects. These differences are also discussed in this chapter.

Some of the information in this section is very technical and complex. At the end of the section, there are recommendations on the best approaches to these issues. These recommendations are intended to make creating the DLL as simple as possible, and to make it easy to use the same source code in LabWindows/CVI and the external compilers.

Calling Convention for Exported Functions

If you intend for your DLL to be used solely by C or C++ programs, you can use the `cdecl` (or WATCOM stack-based) calling convention for your exported functions. If, however, you want your DLL to be callable from environments such as Microsoft Visual Basic, you must declare your exported functions with the `_stdcall` calling convention.

You should do this by explicitly defining the functions with the `_stdcall` keyword. This is true whether or not you choose to make `_stdcall` the default calling convention for your project. You must use the `_stdcall` keyword in the declarations in the include file you distribute with the DLL.

The `__stdcall` keyword in not recognized on other platforms, such as Unix or Windows 3.1. If you are working with source code that might be used on other platforms, you should use a macro in place of `__stdcall`. The `DLLSTDCALL` macro is defined in the `cvidef.h` include file for this purpose.

The following are examples of using the `DLLSTDCALL` macro.

```
int DLLSTDCALL MyIntFunc (void);
char * DLLSTDCALL MyStringFunc (void);
```

Note: ***The*** `stdcall` ***calling convention cannot be used on functions with a variable number of arguments. Consequently, such functions cannot be used in Microsoft Visual Basic.***

Exporting DLL Functions and Variables

When a program uses a DLL, it can access only the functions or variables that are exported by the DLL. Only globally declared functions and variables can be exported. Functions and variables declared as `static` cannot be exported.

If you create your DLL in LabWindows/CVI, there are two ways to indicate which functions and variables to export: the include file method, and the qualifier method.

Include File Method

You can identify symbols to export via include files that contain the declarations of the symbols you want to export. The include files may contain nested `#include` statements, but the declarations in the nested include files are not exported. In the Create Dynamic Link Library dialog box, you select from a list of all of the include files in the project.

The include file method does not work with other compilers. However, it is similar to the `.def` method used by the other compilers.

Export Qualifier Method

You can mark each function and variable you want to export with the an export qualifier. Currently, not all compilers recognize the same export qualifier names. The most commonly used is `__declspec(dllexport)`. Some also recognize `__export`. LabWindows/CVI recognizes both. It is recommended that you use the macro `DLLEXPORT` macro which is defined in the `cvidef.h` include file. The following are examples of using the `DLLEXPORT` macro.

```
int DLLEXPORT DLLSTDCALL MyFunc (int parm) {}
int DLLEXPORT myVar = 0;
```

If the type of your variable or function requires an asterisk (`*`) in the syntax, put the qualifier after the asterisk, as in the following.

```
char * DLLEXPORT myVar = NULL;
```

Note: ***Borland C/C++ version 4.5x, requires that you place the qualifier before the asterisk. In Borland C/C++ 5.0, you can place the qualifier on either side of the asterisk.***

When LabWindows/CVI creates a DLL, it exports all symbols for which export qualifiers appear in either the definition or the declaration. If you use an export qualifier on the definition and an *import* qualifier on the declaration, LabWindows/CVI exports the symbol. The external compilers differ widely in their behavior on this point. Some require that the declaration and definition agree.

Note: ***If you have included in your DLL project an object or library file defining exported symbols, LabWindows/CVI cannot correctly create import libraries for each of the compilers it works with. This problem does not arise if you are using only source code files in your DLL project.***

Marking Imported Symbols in Include File Distributed with DLL

Generally, you should distribute an include file with your DLL. The include file should declare all of the exported symbols. If any of these symbols are variables, you must mark them with an import qualifier. Import qualifiers are *required* on variable declarations so that the correct code can be generated for accessing the variables.

Import qualifiers can also be used on function declarations, but they are not required. When you use an import qualifier on a function declaration, external compilers can generate slightly more efficient code for calling the function.

Using import qualifiers in the include file you distribute with your DLL can cause problems if you use the same include file in the DLL source code.

- If you mark variable declarations in the include file with import qualifiers and you use the include file in a source file other than the one in which the variable is defined, LabWindows/CVI (and any other external compiler) treats the variable as if it were imported from *another* DLL and generates incorrect code as a result.
- If you use export qualifiers in the definition of symbols and the include file contains import qualifiers on the same symbols, some external compilers report an error.

You can solve these problems in several different ways.

- You can avoid exporting variables from DLLs, and thereby eliminate the need to use import qualifiers. For each variable you want to export, you can create functions to get and set its value or a function to return a pointer to the variable. You do not need to use import qualifiers for functions. This is the simplest approach. (Unfortunately, it does not work if you use an export qualifier in a function definition and you are creating the DLL with an external compiler that requires the declaration and definition to agree.)
- You can create a separate include file for distribution with the DLL.
- You can use a special macro that resolves to either an import or export qualifier depending on a conditional compilation flag. In LabWindows/CVI you can set the flag in your DLL project by using the **Compiler Defines** command in the **Options** menu of the Project window.

Recommendations

To make creating a DLL as simple as possible, adhere to the following recommendations.

- Use the `__stdcall` keyword (or `DLLSTDCALL` or a similar macro) in the declaration and definition of all exported functions. Do not export functions with a variable number of arguments.
- Identify the exported symbols using the include file method. Do not use export qualifiers. If you are using an external compiler, use the `.def` file method.
- Do not export variables from the DLL. For each variable you want to export, you can create functions to get and set its value or a function to return a pointer to the variable. Do not use import qualifiers in the include file.

If you follow these recommendations, you reap the following benefits.

- You can distribute with your DLL the same include file that you include in the source files used to make the DLL. This is especially useful when creating DLLs from instrument drivers.
- You can use the same source code to create the DLL in LabWindows/CVI and any of the compatible external compilers.
- You can use your DLL in Microsoft Visual Basic or other non-C environments.

Automatic Inclusion of Type Library Resource for Visual Basic

The **Create Dynamic Link Library** command gives you the option to automatically create a Type Library resource and include it in the DLL. When you use this option, Visual Basic users can call the DLL without having to use a header file containing `Declare` statements for the DLL functions. The command requires that you have a function panel file for your DLL.

If your function panel file contains help text, you can generate a Windows help file from it using the **Generate Windows Help** command in the **Options** menu of the Function Tree Editor . The **Create Dynamic Link Library** command optionally includes pointers into the Window help file in the Type Library. These pointers let Visual Basic users access the help information from the Type Library Browser.

Visual Basic has a more restricted set of types than C. Also, the **Create Dynamic Link Library** command imposes certain requirements on the declaration of the DLL API. Use the following guidelines to ensure that you DLL API can be used in Visual Basic:

- Always use typedefs for structure parameters and union parameters.
- Do not use enum parameters.
- Do not use structures that require forward references or that contain pointers.
- Do not use pointer types except for reference parameters.

Creating Static Libraries in LabWindows/CVI

You can create static library (`.lib`) files in LabWindows/CVI for Windows 95 and NT. Static libraries are libraries in the traditional sense - a collection of object files - as opposed to a dynamic link library or an import library. You can use just one project to create static library files to work with all four compatible external compilers, but only if you include no object or library files in the project.

You need a separate project for each static library you want to create. You must select **Static Library** from the submenu attached to the **Target** command in the **Build** menu of the Project window. When **Static Library** is checkmarked, the **Create Static Library** command appears below the **Target** command in the **Build** menu. Refer to Chapter 2 in this document for detailed information on the **Create Static Library** command.

Note: ***If you include a*** `.lib` ***file in a static library project, all object modules from the*** `.lib` ***are included in the static library. When an executable or DLL is created, only the modules needed from the*** `.lib` ***file are used.***

Creating Object Files in LabWindows/CVI

You can create an object file in LabWindows/CVI by opening a source (`.c`) file and executing the Create Object File command in the **Options** menu of the Source window.

In LabWindows/CVI for Windows 95 and NT, you can choose to create only an object file for the currently selected compiler or to create object files for all four compatible external compilers.

Calling Windows SDK Functions in LabWindows/CVI

You can call Windows SDK Functions in LabWindows/CVI for Windows 95 and NT.

Help for the SDK functions can be obtained by selecting the Windows SDK command in the **Help** menu of any LabWindows/CVI window.

Windows SDK Include Files

You must include the SDK include files *before* the LabWindows/CVI include files. There are function name and typedef conflicts between the Windows SDK and the LabWindows/CVI libraries. The LabWindows/CVI include files contain special macros and conditional compilation to adjust for declarations in the SDK include files. Thus, the C preprocessor must process the SDK include files before the LabWindows/CVI include files.

When you are compiling in LabWindows/CVI or when you are using an external compiler to compile your source files for linking in LabWindows/CVI, use LabWindows/CVI's SDK include files. LabWindows/CVI's SDK include files are in the `cvi\sdk\include` directory. The `cvi\sdk\include` directory is automatically searched by the LabWindows/CVI compiler. You do not need to add it to your include paths.

When you use an external compiler to compile and link your source files, you should use the SDK include files that come with the external compiler. If you use an external compiler to compile your source files for linking in LabWindows/CVI, use LabWindows/CVI's SDK include files. For more information, see the *Setting Up Include Paths for LabWindows/CVI, ANSI C, and SDK Libraries* section later in this chapter.

There are a very large number of SDK include files. In general, you need to explicitly include only `windows.h`. It, in turn, includes many, but not all, of the other include files. Including all of `windows.h` and its subsidiary include files causes a significant increase in compilation time and memory usage. `WIN32_LEAN_AND_MEAN` is a macro from Microsoft which, when defined, eliminates the less commonly used portions of `windows.h` and its subsidiary include

files. By default, LabWindows/CVI adds `/DWIN32_LEAN_AND_MEAN` as a compile-time definition when you create a new project. You can modify this behavior by using the **Compiler Defines** command in the **Options** menu of the Project window.

Using Windows SDK Functions for User Interface Capabilities

The LabWindows/CVI User Interface Library is built on top of the Windows SDK. It is not designed to be used in user programs that attempt to build other user interface objects at the SDK level. While there are no specific restrictions on using SDK functions in LabWindows/CVI, it is recommended that you either use the User Interface Library for your entire user interface, or not use it at all.

Using Windows SDK Functions to Create Multiple Threads

Although you can use the Windows SDK Functions to create multiple threads in a LabWindows/CVI program, the LabWindows/CVI development environment is not set up to handle multiple threads. For instance, if your main program terminates without destroying the threads, they are not terminated. Also, the LabWindows/CVI libraries are not multithreaded safe when called from a program linked in LabWindows/CVI.

(Some of the libraries are multithreaded safe in programs linked with an external compiler. See the *Creating Executables and DLLs in External Compilers for Use with the LabWindows/CVI Libraries* earlier in this chapter.)

Automatic Loading of SDK Import Libraries

All of the SDK functions are implemented in DLLs. Each external compiler comes with a number of DLL import libraries for the SDK functions. Most of the commonly used SDK functions programs are in the following three import libraries.

```
kernel32.lib
gdi32.lib
user32.lib
```

LabWindows/CVI for Windows 95 and NT automatically loads these three libraries at start up and searches them to resolve references at link time. Thus, you do not need to include these libraries in your project.

If the LabWindows/CVI linker reports SDK functions as unresolved references, you need to add import libraries to your project. Refer to the `cvi\sdk\sdkfuncs.txt` file for associations of SDK import libraries to SDK functions. The import libraries are in the `cvi\sdk\lib` directory.

Setting Up Include Paths for LabWindows/CVI, ANSI C, and SDK Libraries

The rules for using SDK include files are not the same as the rules for using ANSI C standard library include files, which in turn are different than the rules for using the LabWindows/CVI library include files. (See the *Include Files for the ANSI C Library and the LabWindows/CVI Libraries* and *Windows SDK Include Files* sections earlier in this chapter.) Depending on where you are compiling and linking, you may have to set up your include paths very carefully. Each of the cases is discussed here.

Compiling in LabWindows/CVI for Linking in LabWindows/CVI

Use the SDK and ANSI C include files that come with LabWindows/CVI. This is done automatically. You do not need to set up any special include paths.

Compiling in LabWindows/CVI for Linking in an External Compiler

Use LabWindows/CVI's SDK include files and the external compiler's ANSI C include files. Using the **Include Paths** command in the **Options** menu of the Project window, add the following as explicit include paths at the beginning of the project-specific list.

```
cvi\include
cvi\sdk\include
directory containing the external compiler's ANSI C include paths
```

Compiling in an External Compiler for Linking in an External Compiler

Use the SDK and ANSI C include files that come with the external compiler. This is done automatically. You do need to specify the `cvi\include` directory as an include path for the LabWindows/CVI library include files.

Compiling in an External Compiler for Linking in LabWindows/CVI

Use the SDK and ANSI C include files that come with LabWindows/CVI. Specify the `cvi\include`, `cvi\include\ansi` and `cvi\sdk\include` directories as include paths in the external compiler.

Run-Time Stack Size

Under LabWindows/CVI for Windows 3.1, your run-time stack is limited to a maximum of 16,384 bytes. (See Table 1-3, *Stack Size Ranges for LabWindows/CV*I, in the *LabWindows/CVI Programmer Reference Manual*.) Under LabWindows/CVI for Windows 95, and NT, the maximum stack size is 1,000,000 bytes. The default size and minimum size is 100,000. You can change the stack size using the **Run Options** command in the **Options** menu of the Project window.

No Floating Point Coprocessor Required

No floating point coprocessor or emulator is required to run LabWindows/CVI for Windows 95 and NT or to use the DLLs containing the LabWindows/CVI libraries for Windows 95 and NT.

New Predefined Macros

The following predefined macros have been added for Windows 95 and NT.

- `_CVI_EXE_` is defined if the project target type is Standalone Executable
- `_CVI_DLL_` is defined if target type is Dynamic Link Library
- `_CVI_LIB_` is defined if target type is Static Library
- `__DEFALIGN` is defined to the default structure alignment (8 for Microsoft and Symantec, 1 for Borland and WATCOM)
- `_NI_VC_` is defined to 220 if in Microsoft Visual C/C++ compatibility mode
- `_NI_SC_` is defined to 720 if in Symantec C/C++ compatibility mode
- `_NI_BC_` is defined to 451 if in Borland C/C++ mode
- `_NI_WC_` is defined to 1050 if in WATCOM C/C++ mode
- `_WINDOWS` is defined

- `WIN32` is defined
- `_WIN32` is defined
- `__WIN32__` is defined
- `__NT__` defined
- `_M_IX86` is defined to 400
- `_NI_mswin32_` is defined

General Compiler/Linker Enhancements

The compiler/linker enhancements made specifically for Windows 95 and NT are discussed earlier in this chapter. This section discusses other changes made to the LabWindows/CVI compiler and linker. Unless otherwise marked, these changes apply to LabWindows/CVI on all platforms.

Maximum Nesting of Include Files

The maximum nesting of `#include` statements has been increased from 8 to 32. (For other limits, see Table 1-1, *LabWindows/CVI Compiler Limits*, in the *LabWindows/CVI Programmer Reference Manual*.)

C Language Extensions

Several extensions to, or relaxations of, the C language have been made. The purpose is to make the LabWindows/CVI compiler compatible with the commonly used C extensions in external compilers on Windows 95 and NT.

Calling Conventions (Windows 95/NT Only)

You may use the following calling convention qualifiers in function declarations:

`cdecl`
`_cdecl`
`__cdecl` (recommended)
`_stdcall`
`__stdcall` (recommended)

In Microsoft Visual C/C++, Borland C/C++, and Symantec C/C++, if you do not use a calling convention qualifier, the calling convention normally defaults to `cdecl`. You can, however, set options to cause the calling convention to default to `stdcall`. The behavior is the same in LabWindows/CVI. You can set the default calling convention to either `cdecl` or `stdcall` using the **Compiler Options** command in the **Options** menu of the Project window. When you create a new project, the default calling convention is `cdecl`.

In WATCOM C/C++, the default calling convention is neither `cdecl` nor `stdcall`. When you compile a module in WATCOM for use in LabWindows/CVI, you must use the `-4s` (80486 Stack-Based Calling) option. (See the *Creating Object and Library Files in External Compilers for Use in LabWindows/CVI* section in this chapter.) The `-4s` option causes the stack-based

calling convention to be the default. In LabWindows/CVI in WATCOM compatibility mode, the default calling convention is always the stack-based convention. It cannot be changed. LabWindows/CVI does compile with the `cdecl` and `stdcall` conventions under WATCOM, except that floating point and structure return values do not work in the `cdecl` calling convention. It is recommended that you avoid using `cdecl` with WATCOM.

In the `cdecl` calling convention (and the WATCOM stack-based calling convention), the calling function is responsible for cleaning up the stack, and functions can have variable number of arguments.

In the `stdcall` calling convention, the called function is responsible for cleaning up the stack. Functions with a variable number of arguments do not work in `stdcall`. If you use the `stdcall` qualifier on a function with a variable number of arguments, the qualifier is not honored. All compilers pass parameters and return values in the same way for `stdcall` functions, except for floating point and structure return values.

The `stdcall` calling convention is recommended for all functions exported from a DLL. Visual Basic and other non-C Windows programs expect DLL functions to be `stdcall`.

Import and Export Qualifiers

You may use the following qualifiers in variable and function declarations.

```
__declspec(dllimport)
__declspec(dllexport)
__import
__export
_import
_export
```

At this time, not all of these qualifiers work in all external compilers. The LabWindows/CVI `cvidef.h` include file defines the following two macros, which are guaranteed to work in each external compiler.

```
DLLIMPORT
DLLEXPORT
```

An import qualifier informs the compiler that the symbol is not defined in the project but rather in a DLL that is linked into the project. Import qualifiers are required on declarations of variables imported from a DLL, but are not required on function declarations.

An export qualifier is relevant only in a project for which the target type is Dynamic Link Library. The qualifier can be on the declaration or definition or symbol, or both. The symbol must be defined in the project. The qualifier instructs the linker to include the symbol in the DLL import library.

C++-Style Comment Markers

You can use double slashes (//) to begin a comment. The comment continues until the end of the line.

Duplicate Typedefs

The LabWindows/CVI compiler no longer reports an error on multiple definitions of the same `typedef` identifier, as long as the definitions are identical.

Structure Packing Pragma (Windows 3.1 and Windows 95/NT only)

The `pack pragma` now works in LabWindows/CVI. You can use it to specify the maximum alignment factor for elements within a structure. For example, assume the following structure definition,

```
struct t {
   double d1;
   char charVal;
   short shortVal;
   double d2;
  };
```

If the maximum alignment is 1, the structure can start on any 1-byte boundary, and there are no gaps between the structure elements.

If the maximum alignment is 8, then this structure must start on an 8-byte boundary, `shortVal` starts on a 2-byte boundary, and `d2` starts on an 8-byte boundary.

You can set the maximum alignment as follows:

```
#pragma pack(4) /* sets maximum alignment to 4 bytes */
#pragma pack(8) /* sets maximum alignment to 8-bytes */
#pragma pack() /* resets to the default */
```

The maximum alignment applied to a structure is based on the last `pack pragma` statement seen before the definition of the structure.

Program Entry Points (Windows 95 and NT only)

In Windows 95 and NT, you can use `WinMain` instead of `main` as the entry point function to your program. You might want to do this if you plan to link your executable using an external compiler. You need to include `windows.h` for the data types normally used in the `WinMain` parameter list. The following is the prototype for `WinMain` with the Windows data types reduced to intrinsic C types.

```
int __stdcall WinMain(void * hInstance, void * hPrevInstance, char * lpszCmdLine
                      int nCmdShow)
```

Include Paths

Several changes have been made in how LabWindows/CVI searches for include files.

Non-Project-Specific User-Defined Include Paths

In previous versions of LabWindows/CVI, the include paths you specified using the **Include Paths** command in the **Options** menu of the Project window were always kept in the project file. If you shared a project file with other users, the include paths were carried along with it, even though the directories on the other machines may have been structured very differently.

Now you can specify include paths to be kept for your machine only, without regard to the project file. The **Include Paths** dialog box has two lists, one for include paths specific to the project, and one not specific to the project.

VXI Plug & Play Include Directory

When you install VXI Plug & Play instrument drivers, the include files for the drivers are placed in a specific VXI Plug & Play include directory. LabWindows/CVI now searches that directory for include files.

Complete Search Precedence

The following is the complete include file search precedence used by LabWindows/CVI.

- Project list
- Project-specific user-defined include paths

- Non-project-specific user-defined include paths
- The paths listed in the Instrument Directories dialog box
- The `cvi\include` directory
- The `cvi\include\ansi` directory
- The VXI*plug&play* include directory
- The `cvi\instr` directory
- The `cvi\include\sdk` directory (Windows 95/NT only)

Searching for Instrument Driver DLLs (Windows 3.1 Only)

This section describes a documentation correction and an enhancement made regarding how instrument drivers DLLs are found in LabWindows/CVI for Windows 3.1.

Correction to Documentation

There is an error in the *DLL Search Precedence* section of Chapter 2, *Using Loadable Compiled Modules*, in the *LabWindows/CVI Programmer Reference Manual*. It states that if a DLL is associated with an `.fp` file, LabWindows/CVI looks in the project for a `.pth` or `.dll` file with the same base name as the `.fp` file. In fact, it looks in the project only for a `.pth` or `.dll` file with the same *full* path name as the `.fp` file, except for the extension.

Searching for DLLs Associated with .fp Files

Starting in LabWindows/CVI version 4.0 for Windows 3.1, if there is not a `.pth` or `.dll` file in the same directory as the `.fp` file, LabWindows/CVI looks for a DLL with the same base name as the `.fp` file using the standard Windows search algorithm. Thus, if a DLL with the same base name is in the `windows` or `windows/system` directory or a directory listed in your `PATH` environment variable, LabWindows/CVI finds it.

This makes it easier to use VXI Plug & Play instrument driver DLLs in LabWindows/CVI for Windows 3.1. DLLs for VXI Plug & Play drivers are not in the same directory as the .fp files, but the directory containing the DLL is listed in the `PATH` environment variable.

Note: ***In Windows 95 and NT, LabWindows/CVI never directly searches for DLLs associated with*** `.fp` ***files. Each DLL must have a DLL import library (***`.lib`***) file. The DLL import library specifies the name of the DLL, which is then searched for using the standard Windows DLL search algorithm. If the*** `.fp` ***file is under the VXIplug&play directory tree, LabWindows/CVI looks for a*** `.lib` ***file in the VXIplug&play import***

library subdirectory. LabWindows/CVI looks for the VXIplug&play import library before it looks for a program file in the directory of the `.fp` ***file, unless a program file in the directory of the*** `.fp` ***file (and with the same base name as the*** `.fp` ***file) is listed in the project and is unexcluded.***

Run State Change Callbacks - Clarification

When you use a compiled module in LabWindows/CVI, you can arrange for it to be notified of a change in the execution status (start, stop, suspend, resume). This is done through a callback function, which is always named `__RunStateChangeCallback`. This is described in detail in the section *Special Considerations When Using a Loadable Compiled Module*, in Chapter 2, *Using Loadable Compiled Modules*, of the *LabWindows/CVI Programmer Reference Manual*.

The description implies that a `Suspend` notification is always followed by a `Resume` notification. In actuality, however, a `Stop` notification can follow a `Suspend` notification without an intervening `Resume` notification.

Note: ***Run State Change Callbacks do not work in programs linked in external compilers.***

Distributing Executables, DLLs, and Libraries in Windows 95

This chapter discusses distributing standalone executables and DLLs in Windows 95 and NT. Chapter 4, *Creating and Distributing Standalone Executables*, of the *LabWindows/CVI Programmer Reference Manual*, describes this process for Windows 3.1. The process for Windows 95 and NT is very similar. This chapter follows the organization of Chapter 4 of the *LabWindows/CVI Programmer Reference Manual*, noting the differences and changes for Windows 95 and NT.

One section discusses using hardware interrupts in Windows 95 and NT. Also, new compiler, linker, and run-time error messages are presented. Changes to Chapter 5, *Distributing Libraries*, in the *Programmers Reference Manual*, are also described.

The Run-Time Library DLLs

For Windows 3.1, executables that you distribute must be accompanied by the LabWindows/CVI run-time engine, which is an executable file. The run-time engine is distributed with LabWindows/CVI on a separate diskette and is installed as part of the LabWindow/CVI installation. The **Create Distribution Kit** command in the **Build** menu of the Project window optionally bundles the run-time engine into your distribution kit. Alternatively, you can make copies of this diskette for separate distribution.

For Windows 95 and NT, the run-time libraries are in a set of DLLs rather than in an executable file. As in Windows 3.1, these DLLs are distributed on a separate diskette and are installed as part of LabWindows/CVI. The **Create Distribution Kit** command in the **Build** menu of the Project window optionally bundles the run-time library DLLs into your distribution kit. Alternatively, you can make copies of this diskette for separate distribution.

The run-time library DLLs are the following.

```
cvirt.dll
cvirte.dll
```

The LabWindows/CVI run-time library DLLs do not include the DLLs for National Instruments hardware. End-users can install the DLLs for their hardware from the distribution disks that National Instruments supplies to those users.

Distributing DLLs You Create

In Windows 95 and NT, you can distribute DLLs that use the LabWindows/CVI run-time libraries. As in the case of standalone executables, they must be distributed along with the LabWindows/CVI run-time library DLLs.

Minimum System Requirements

To use a standalone executable or DLL that depends on the LabWindows/CVI run-time libraries, you must have the following:

- Windows 95, or Windows NT version 3.51 or later
- A personal computer using at least a 33 MHz 80486 or higher microprocessor
- A VGA resolution (or higher) video adapter
- A minimum of 8 MB of memory
- Free hard disk space equal to 4 MB, plus the size of your executable or DLL, plus the size of any files it needs

No Math Coprocessor Required

Unlike the LabWindows/CVI run-time engine for Windows 3.1, you do not need a math coprocessor or emulator to use the LabWindow/CVI run-time libraries in Windows 95 or NT.

Configuring the Run-Time Library DLLs

The options for configuring the run-time library DLLs are the same as those described in the *Configuring the Run-Time Engine* section in Chapter 4, *Creating and Distributing Standalone Executables*, of the *LabWindows/CVI Programmer Reference Manual*.

In Windows 3.1, the configuration options are stored in the `win.ini` file. In Windows 95 and NT, the configuration options are stored in the Registry under the following key.

```
HKEY_LOCAL_MACHINE\Software\National Instruments\CVI Run-time Engine
```

Location of Files on the Target Machine

The *Location of Files on the Target Machine for Running Executable Programs* section in Chapter 4, *Creating and Distributing Standalone Executables*, of the *LabWindows/CVI*

Programmer Reference Manual, was written for the Windows 3.1 version of LabWindows/CVI. Although the section is generally applicable to Windows 95 and NT, different rules apply for locating DLLs used by the executable or DLL you are distributing. (In this manual, DLLs called by your executable or DLL are called subsidiary DLLs).

In Windows 95 or NT, subsidiary DLLs cannot be referenced directly in your project, and DLL path (`.pth`) files are not supported. Your executable or DLL can link to a subsidiary DLL only via an import library. An import library can be linked into your program in any of the following ways.

- It can be listed in your project.
- It can be the program file associated with the `.fp` file for an instrument driver or user library.
- It can be dynamically loaded via a call to `LoadExternalModule`.

Rules for Using Statically Linked DLL Files

If a DLL import library is listed in the project or is associated with an instrument driver or user library, the import library is statically linked into your executable or DLL. The import library contains the name of the subsidiary DLL. When your executable or DLL is loaded, the subsidiary DLL is found using the standard Windows DLL search algorithm, which is described in the Windows SDK documentation for the `LoadLibrary` function. The search precedence is:

- The directory from which the application was loaded
- The current working directory
- On Windows 95, the Windows `system` directory. On Windows NT, the Windows `system32` and `system` directories
- The Windows directory
- The directories listed in the `PATH` environment variable

Rules for Loading Files Using LoadExternalModule

Under Windows 95 and NT, the rules for loading files using `LoadExternalModule` are the same as for Windows 3.1, with the following exceptions.

- DLL path (`.pth`) files do not work.
- DLL files cannot be directly referenced in calls to `LoadExternalModule`. The DLL import library must be referenced instead. The DLL import library contains the name of the DLL file. When `LoadExternalModule` is called from either an executable or a DLL, the DLL file specified in the import library is found using the standard Windows DLL search

algorithm, which is described in the Windows SDK documentation for the `LoadLibrary` function. The search precedence is:

- The directory from which the application was loaded
- The current working directory
- On Windows 95, the Windows `system` directory. On Windows NT, the Windows `system32` and `system` directories
- The Windows directory
- The directories listed in the `PATH` environment variable

Distributing Libraries in Windows 95 and NT

In general, the information in Chapter 5, *Distributing Libraries*, in the *Programmers Reference Manual*, applies to Windows 95 and NT. There is one exception. The *Adding Libraries to User's Library Menu* section describes how you can insert your libraries into the user's library menu by using the `modini` program to modify the user's `cvi.ini` file. In Windows 95 and NT, there is no `cvi.ini` file. Instead, the configuration information for the LabWindows/CVI development environment is kept in the Windows registry. You can use the new program, `modreg`, to modify any information in the registry. Documentation for the program can be found in the file `modreg.doc`. Both `modreg.exe` and `modreg.doc` are in the `cvi\bin` directory.

The `modreg` commands to add files to the user's library menu are same as the `modini` commands shown in the manual. However, you must add the following to the beginning of the `modreg` command file.

```
setkey   [HKEY_CURRENT_USER\Software\National Instruments]
appendkey  CVI\@latestVersion
```

Handling Hardware Interrupts under Windows 95 and NT

In Windows 3.1, you can handle hardware interrupts in a DLL. In Windows 95, you must handle hardware interrupts in a VxD. In Windows NT, you must handle hardware interrupts in a kernel mode driver. VxDs and kernel mode drivers cannot be created in LabWindows/CVI. Instead, they must be created in Microsoft Visual C/C++, and you must have the Microsoft Device Driver Developer Kit (DDK).

In Windows 3.1, it is extremely difficult for you to call into main application source code at interrupt time. Windows 95 and NT make this easier. You can arrange for a function in your LabWindows/CVI source code to be called after your VxD (or kernel mode driver) interrupt service routine exits. You do this by creating a thread for your interrupt callback function. The callback function executes a loop which blocks its thread until it is signaled by the interrupt service routine. Each time the interrupt service routine executes, it unblocks the callback thread. The callback thread then performs its processing and blocks again.

LabWindows/CVI includes source code template files for a VxD and a kernel mode driver. It also includes a sample main program to show you how to read and write registers on a board. We have one set of files for Windows 95 and another for Windows NT.

The files are located in `cvi\vxd\win95` and `cvi\vxi\winnt`. Some basic information is contained in the file `template.doc` in each directory.

New Compiler/Linker/Run-Time Errors and Warnings

This chapter contains the error and warnings messages that should be added to Appendix A, *Errors and Warnings*, in *the LabWindows/CVI Programmers Reference Manual*. These messages are displayed on errors or warning that occur when compiling, linking, or running programs.

Table 1-1. Error Messages for Appendix A, Programmer Reference Manual

Error Message	Type Error	Comment
`Bad BSS section encountered while reading external module: FILE.`	Object Load Error	The object module is corrupted or is of a type that cannot be loaded into LabWindows/CVI.
`Bad COFF Library header.`	Object Load Error	The library file you are loading is either corrupted or not in the COFF format.
`Bad COFF Library member header.`	Object Load Error	The COFF library you are loading contains a module that is corrupted or in an invalid format.
`Cannot link variable '%s' to import library without '%s' keyword in declaration.`	Link Error	A variable that you have declared as `extern` is defined in a DLL import library, but you did include the `__import` or `declspec(dllimport)` qualifier in the declaration.
`COFF Name too long.`	Object Load Error	The COFF object or library you are loading contains a symbol name that is longer than the maximum legal length.
`Could not allocate stack space. Try decreasing the Maximum stack size option in the Run Options dialog.`	Fatal Runtime Error	There is insufficient memory to allocate the Maximum Stack Size you have specified. LabWindows/CVI allocates the maximum size on the stack at the beginning of execution.

continues

Table 1-1. Error Messages for Appendix A, Programmer Reference Manual (Continued)

Error Message	**Type Error**	**Comment**
`Elf library is out of date.`	Object Load Error	CVI expects a more recent version of the shared library (`libelf.so`) that it uses to load ELF objects. As a result, LabWindows/CVI is unable to read or write object and library files.
`Error in Elf Library encountered while reading external module: NAME.`	Object Load Error	The object module is corrupted or is of a type that cannot be loaded by LabWindows/CVI.
`Error: compiling ‘%s’ for DLL exports.`	DLL Import Library Creation Error.	When creating a DLL using the Include File method for specifying exported symbols, an error was encountered compiling the include file.
`Error: Incompatible type for function or variable "%s" in header "%s" used to specify exports.`	DLL Link Error	When creating a DLL using the Include File method for specifying exported symbols, the type of the symbol in the include file does not match the type in the source file.
`Expecting integer constant, push or pop.`	Compile Error	The pack pragma requires at least one parameter.
`Illegal type for symbol 'DllMain': TYPE.`	Compile Error	The function `DLLMain` does not conform to the accepted prototype.
`Illegal type for symbol 'WinMain': TYPE.`	Compile Error	The function `WinMain` does not conform to the accepted prototype.
`Import Variables cannot be used in global variable initialization.`	Compile Error	A global variable marked as `__import` or `declspec(dllimport)` is being used in an initializer of another variable.
`Insufficient system memory for Interactive Window`	Link Error	There is not enough memory to run the interactive window.
`Insufficient system memory for project.`	Link Error	There is not enough memory to link the project.
`Insufficient user data memory for project.`	Link Error	There is not enough memory to link the project.

continues

Table 1-1. Error Messages for Appendix A, Programmer Reference Manual (Continued)

Error Message	Type Error	Comment
`Matching push not encountered or already popped.`	Compile Error	A pack pragma used a named pop that does not balance with push of the same name.
`naked functions are not supported.`	Compile Error	LabWindows/CVI does not work with the `naked` keyword.
`No pack settings currently pushed.`	Compile Error	A pack pragma used a pop when there were no pushes.
`Object module contains unsupported FAR pointers.`	Object Load Error	The external object module contains FAR pointers, which you cannot implement in LabWindows/CVI.
`Pack pragma valid values are 1, 2, 4, 8, and 16.`	Compile Error	The pack pragma alignment value parameter is limited to 1, 2, 4, 8, or 16.
`pragma pack(pop...) does not set alignment. Use separate pack pragma.`	Compile Warning	A pragma pop was used with an alignment value. Use separate pack pragmas for popping and setting the alignment value.
`Symbol ‘%s’ exported from header "%s" not found in DLL.`	DLL Link Error or Import Library Creation Error.	When creating a DLL using the Include File method for specifying exported symbols, one of the symbols declared in the include file was not found in the DLL project. Or, when creating import libraries from an include file and a DLL, one of the symbols declared in the include file was not found in the DLL.
`Syntax error; found TOKEN1 expecting TOKEN2.`	Compile Error	A syntax error occurred because TOKEN1 was found instead of TOKEN2.
`The __cdecl calling convention is not supported with functions returning floats, doubles, or structures in WATCOM Compatibility Mode.`	Compile Error	A function with an explicit `__cdecl` qualifier returns a `double`, `float` or structure, and the selected compatible compiler is WATCOM. Either remove the qualifier or change the function.

continues

Table 1-1. Error Messages for Appendix A, Programmer Reference Manual (Continued)

Error Message	**Type Error**	**Comment**
`The callback function, NAME, differs only by a leading underscore from another function or variable. Change one of the names for proper linking.`	Non-Fatal Runtime Error	When trying to match a callback name specified in a `.uir` file to the callback function, the compiler found two symbols that are the same except for a leading underscore. Resolve this ambiguity by changing the one of the names.
`Thread data is not supported.`	Compile Error	You cannot implement thread-local storage in LabWindows/CVI.
`Type error: pointer expected.`	Compile Error	The expression being dereferenced with the '`*`', '`->`' or '`[]`' operator does not have pointer type.
`Unnamed pop matching named push.`	Compile Warning	A pack pragma used a unnamed pop that balances a name push.
`Warning: Import libraries other than the one for the current compatibility mode may not work for symbols exported from an object file. It is recommended that you export using header files instead.`	DLL Link Warning	When creating a DLL using the Symbols Marked for Export method for specifying exported symbols, one of the modules was an object or library file. LabWindows/CVI does not have sufficient information to ensure that the import libraries it generates for all four compatible external compilers will have the correct names of the symbols in that module.

Chapter 2
Updates to the User Manual

Chapter Contents

Project Window Changes

This chapter discusses the changes to the Project window, including the menu commands accessible from the Project window. This chapter follows the organization of Chapter 3, *Project Window*, of the *User Manual*. The changes apply to all platforms, unless otherwise marked.

File Menu

This section discusses the changes to the **File** menu in the Project window.

Auto Save Project

The **Auto Save Project** command has replaced the **Read Only** command in the **File** menu of the Project Window. When a project is loaded, the **Auto Save Project** command is initially enabled unless the project file is read-only on disk. If the command is enabled, the LabWindows/CVI automatically saves the project file whenever there is significant new or modified information to save in the project. If the command is disabled, the project file is saved only in the following cases.

- The **Save**, **Save As**, or **Save All** command is executed from the **File** menu.
- You unload the project or exit LabWindows/CVI. (You are prompted to save the file in this case).

Notice that if the **Auto Save Project** command is disabled, the project file is not saved when you start running a program, even if the **Save changes before running** option in the Run Options dialog box is set to **Always** or **Ask**.

Print

A **Print** command has been added to the **File** menu in the Project window. It brings up a selectable list of all of the files in the project that are printable. You can checkmark the files you want to print.

Most Recently Closed Files

The **File** menu now contains two lists.

- a list of the four most recently closed files (other than project files)
- a list of the four most recently closed project files

Edit Menu

This section discusses the changes to the **Edit** menu in the Project window.

Use Import Libraries in Project Instead of .dll and .pth Files (Windows 95/NT Only)

In Windows 95 and NT, each DLL must be accompanied by an import library (`.lib`) file. If you want to use a DLL in your project, you must list the import library rather than the DLL. DLL and DLL path (`.pth`) files cannot be added to the project.

If you load a project that was created in Windows 3.1 and that contains `.dll` or `.pth` files, a warning message appears, and the files are excluded.

For more detailed information on using DLLs in LabWindows/CVI for Windows 95 and NT, see the *Loading DLLs in LabWindows/CVI* section in Chapter 1, *Updates to the Programmer Reference Manual* in this document.

Build Menu

This section discusses the changes to the **Build** menu in the Project window.

Target (Windows 95/NT Only)

The **Target** item brings up a submenu in which you select the target type for your project. The target type determines what type of file is created when you execute the command that appears below **Target** in the **Build** menu. The command that appears below **Target** in the **Build** menu changes name depending on the target type selected. The target types that you can select are:

- Standalone Executable
- Dynamic Link Library
- Static Library

When anything other than **Standalone Executable** is selected, the **Build Project** command in the **Build** menu and the **Run Project** command in the **Run** menu are dimmed.

External Compiler Support (Windows 95/NT only)

Use the **External Compiler Support** command to help you build your executable or DLL in one of the four compatible external compilers. For detailed information on this topic, see the *Creating Executables and DLLs in External Compilers for Use with the LabWindows/CVI Libraries* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document.

When you execute the command, the External Compiler Support dialog box appears, as shown in the following figure.

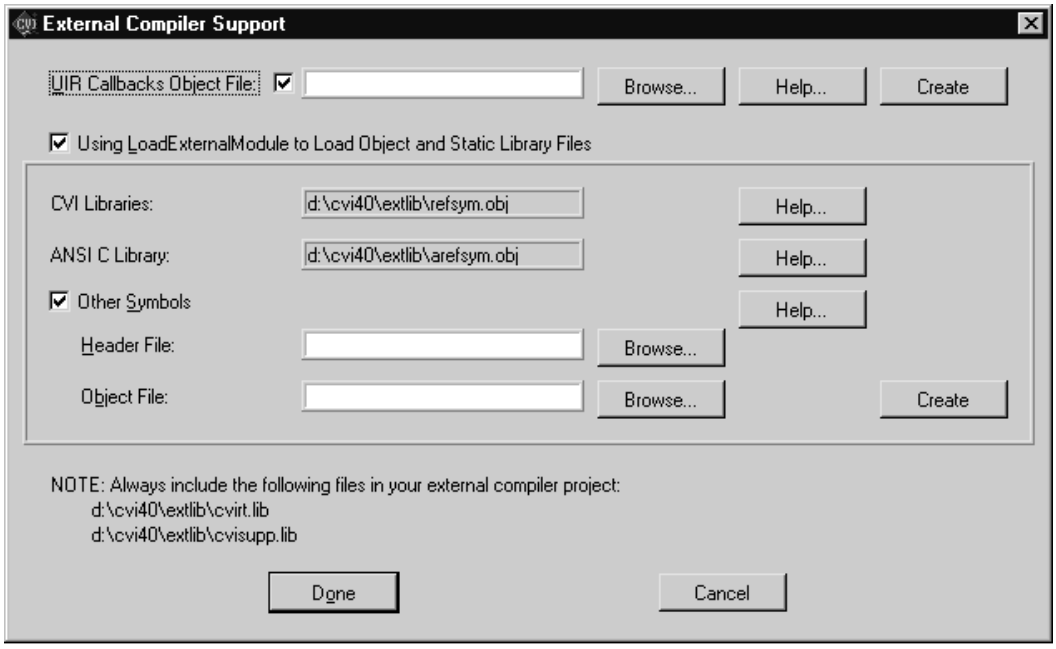


Figure 2-1. External Compiler Support Dialog Box

- **UIR Callbacks Object File**—This option creates an object file for you to link into your executable or DLL. The object file contains a list of the callback functions specified in the User Interface Resource (`.uir`) files in your project. When you load a panel or menu bar from the `.uir` file, the User Interface Library uses the list to link the objects in the panel or menu bar to their callback functions in your executable or DLL. If you specify callback function names in your `.uir` file(s), checkmark the checkbox, enter the name of the object file to be created, and click on the **Create** button. In the future, whenever you save modifications to any of the `.uir` files in the project, LabWindows/CVI automatically updates the object file.

  You must call the `InitCVIRTE` function at the beginning of your `main`, `WinMain`, or `DLLmain` function so that LabWindows/CVI run-time libraries can initialize the list of names from the object file. If you are creating a DLL and any of your callback functions are

defined in, but not exported by, the DLL, you must call `LoadPanelEx` or `LoadMenuBarEx` (rather than `LoadPanel` or `LoadMenuBar`) from the DLL.

- **Using LoadExternalModule to Load Object and Static Library Files**—This option enables the section of the dialog box that you use when creating an executable or DLL that calls the Utility Library `LoadExternalModule` function to load object or static library files.

  Note: ***You do not need this option if you use*** `LoadExternalModule` ***to load only DLLs (which are loaded via DLL import libraries).***

  Unlike DLLs, in which all of the external references are resolved at link time, objects and static libraries can contain unresolved external references. When you use `LoadExternalModule` to load an object or static library file, these references are resolved using symbols in your executable or DLL, or in previously loaded external modules. Consequently, the names of the symbols in your executable or DLL that are needed to resolve these references must be available to the `LoadExternalModule` function.

- **CVI Libraries**—Checkmark this option if your run-time modules reference symbols in any of the following LabWindows/CVI libraries:
  - User Interface
  - RS-232
  - DDE
  - TCP
  - Formatting and I/O
  - Utility

  If you use one of these libraries, include in your external compiler project the object file displayed in this option.

- **ANSI C Library**—Checkmark this option if your run-time modules reference symbols in the ANSI C library. Include in your external compiler project the object file displayed in this option.

- **Other Symbols**—Checkmark this option if your run-time modules reference symbols other than those covered by the previous two options. Such symbols include functions or variables that are defined globally in your executable or DLL and to which your object or static library run-time modules expect to link. This option creates a file for you to link into your executable (or DLL).
  - **Header File**—Insert the name of an include file that contains complete declarations of all of the symbols needed to resolve references from run-time modules.
  - **Object File**—Enter the name of the object file that is to be created. Click on the **Create** button to create the file. You must include this file in your external compiler project.

Create Standalone Executable

The following new item appears in the Create Standalone Executable dialog box for Windows 95 and NT.

- **Version Info**—When you click on this button the Version Info dialog box appears, where you can enter version information for the executable file. The information is saved in the executable in the form of a standard Windows version resource. The information can be obtained from the executable by using the Windows SDK functions `GetFileVersionInfo` and `GetFileVersionInfoSize`.

In the Version Info dialog box, the entries for **File Version** and **Product Version** must be in the form,

`n,n,n,n`

where n is a number from 0 to 255

Create Dynamic Link Library (Windows 95/NT Only)

Use the **Create Dynamic Link Library** command to create a dynamic link library (`.dll`) file from the current project. A DLL import library (`.lib`) file is also created. When you select the command, the Create Dynamic Link Library dialog box appears as shown in the following figure.

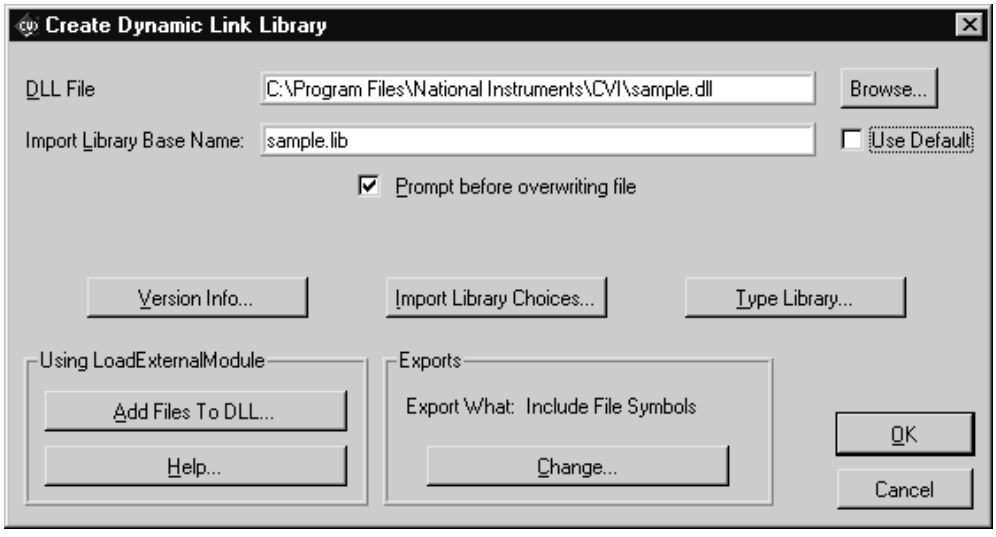


Figure 2-2. The Create Dynamic Link Library Dialog Box

- **DLL File**—The name of the DLL file to be created. You can use the **Browse** button to select an existing filename or enter a new one.

- **DLL Import Library Base Name** — Normally the name of the import library is the same as the name of the DLL, except that the extension is `.lib`. There may be some cases, however, where want to use a different name. For example, you may want to append "`_32`" to the name of your DLL to distinguish it as a 32-bit DLL, but not append it to the import library name. This is, in fact, the convention used for VXI Plug & Play instrument driver DLLs. If you want to enter a different name for the import library, deselect the **Use Default** checkbox. Enter a name without any directory names and without an extension.
- **Prompt before overwriting file**—A checkbox where you can choose to be prompted before the program overwrites a DLL file that has the same name as the one you are creating.
- **Version Info**—When you click on this button the Version Info dialog box appears, where you can enter version information for the DLL. The version information is saved in the DLL in the form of a standard Windows version resource. The information can be obtained from the DLL by using the Windows SDK functions `GetFileVersionInfo` and `GetFileVersionInfoSize`.

  In the Version Info dialog box, **File Version** and **Product Version** must be in the form,

  `n,n,n,n`

  where n is a number from 0 to 255
- **Import Library Choices**—This button lets you choose whether to create a DLL import library for each of the compatible external compilers, or to create one only for the current compatible compiler. (See the *Compatibility with External Compilers* section in Chapter 1 *Updates to the Programmer Reference Manual*, of this document.) If you choose to create an import library for each compiler, the files are created in subdirectories named `MSVC`, `BORLAND`, `WATCOM`, and `SYMANTEC`. The library for the current compatible compiler is also created in the directory of the DLL.
- **Type Library**—This button lets you choose whether to add a Type Library resource to your DLL. This feature is useful if you intend your DLL to be used from Visual Basic. For more information, see the section *Automatic Inclusion of Type Library Resource for Visual Basic* section in Chapter 1, *Updates to the Programmer Reference Manual* in this document.
- **Using LoadExternalModule**
  - **Add Files to DLL**—This button lets you select additional module files which you want to be linked into the DLL. These are modules which are not directly referenced by your project files but which are referenced by modules you load at run-time by calling `LoadExternalModule`.
  - **Help**—This button describes the use of `LoadExternalModule` in a DLL and the **Add Files to DLL** button.
- **Exports**
  - **Export What**—This indicates your current choice of method for determining which symbols in the DLL are exported to the users of the DLL. The **Change** button is used to change your choice.

– **Change**—This button lets you select the method to use for determining which symbols in the DLL are exported to the users of the DLL. The choices are the following.

Include File Symbols—You must name one or more include files that declare symbols defined globally in the DLL. The declared symbols are the ones exported. You can select from a list of include files in the project.

Symbols Marked for Export—All symbols in the DLL defined with qualifier `__declspec(dllexport)` or `export` are exported.

- **OK**—This button accepts the current inputs and attempts to create the DLL.
- **Cancel**—This button cancels the operation and removes the dialog box.

Note: ***When you use the* Symbols Marked for Export *option and have included in your project an object or library file defining exported symbols, LabWindows/CVI cannot correctly create the import libraries for each of the four compatible compiler. This problem does not arise if you are using only source code files in your DLL project.***

For more information on creating DLLs, see the *Preparing Source Code for Use in a DLL* section in Chapter 1, *Updates to the Programmer Reference Manual* in this document.

Create Static Library (Windows 95/NT Only)

Use this **Create Static Library** command to create a static library (`.lib`) file from the current project. When you select the command, the Create Static Library dialog box appears as shown in the following figure.

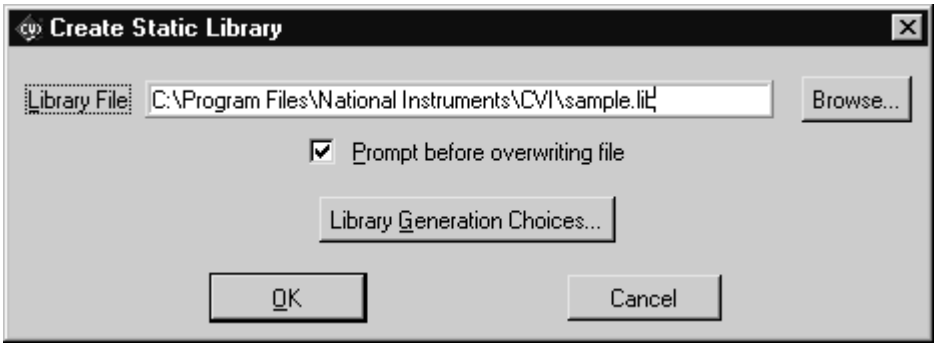


Figure 2-3. The Create Static Library Dialog Box

- **Library File—**The name of the library file to be created. You can use the **Browse** button to select an existing filename or name a new one.
- **Prompt before overwriting file**—A checkbox where you can choose to be prompted before the program overwrites a library file that has the same name as the one you are creating.

- **Library Generation Choices**—This button lets you choose whether to create a static library for each of the compatible external compilers, or to create one only for the current compatible compiler. (See the *Compatibility with External Compilers* section in Chapter 1, *Updates to the Programmer Reference Manual*, of this document.) If you want to create a static library for each compiler, you must not include any object or library files in your project, because such files are specific to particular compiler.
- If you choose to create a static library for each compiler, the files are created in subdirectories named `MSVC`, `BORLAND`, `WATCOM`, and `SYMANTEC`. The library for the current compatible compiler is also created in the parent directory.
- **OK**—This button accepts the current inputs and attempts to create the library file.
- **Cancel**—This button cancels the operation and removes the dialog box.

Note: ***If you include a*** `.lib` ***file in a static library project, all object modules from the*** `.lib` ***are included in the static library. When an executable or DLL is created, only the*** `.lib` ***modules referenced by other modules in the project are included in the target. In addition, LabWindows/CVI reports an error if you attempt to build a static library when you have a DLL import library in your project.***

Create Distribution Kit (Windows 3.1 and Windows 95/NT Only)

The Create Distribution Kit command now includes an uninstall program on the distribution disk.

Another choice has been added to the **Replace Existing Files** option. The **Check Version** option applies to files with a Windows version resource (in other words, DLLs and executables). The file on the distribution kit and the existing file are checked for a version resource. If each has a version resource, the existing file is replaced if the version number of the file in the distribution kit is newer than the version number in the existing file. If neither files have a version resource, the existing file is replaced if its date is newer. If only one file has a version resource, it is considered to be the newer one.

The following new options appear in the Create Distribution Kit dialog box.

- **Distribute Objects and Libraries for All Compilers—**This checkbox appears in LabWindows/CVI for Windows 95 and NT to help you distribute object files, static libraries, and DLL import libraries for all of the compatible external compilers. When selected, this option affects all of the `.obj` and `.lib` files listed in the current file group. Four versions of each file are included in the distribution kit. These versions are expected to be in subdirectories under the specified location of each file. The subdirectories must be named `MSVC`, `BORLAND`, `WATCOM`, and `SYMANTEC`. For example, if you specify the following file,

  ```
  c:\myapp\distr\big.lib
  ```

in a file group and the **Distribute Objects and Libraries for All Compilers** checkbox is checked, then when the distribution kit is created, you must have the following files on your disk:

```
c:\myapp\distr\msvc\big.lib
c:\myapp\distr\borland\big.lib
c:\myapp\distr\watcom\big.lib
c:\myapp\distr\symantec\big.lib
```

When installing the software, the end-user is prompted to choose one of the compatible external compilers. Only the versions of the files for the chosen compiler are actually installed on the end-user's computer.

You might want to use this feature if you are distributing modules for use with the LabWindows/CVI development environment or external compilers. If you are distributing a turnkey application, you do not need this feature.

- **Install Low-Level Support Driver**—This option appears in Windows 95 and NT. It lets you choose whether to install the LabWindows/CVI low-level support driver on the end-user's computer. The following Utility Library functions require the LabWindows/CVI low-level driver to be loaded at startup.

Table 2-1. Platforms Where Utility Functions Need Low-Level Support Driver

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

- **Advanced**—When you click on this button, the Advanced Distribution Kit Options dialog box appears.

Advanced Distribution Kit Options

Clicking on the **Advanced** button in the Create Distribution Kit dialog box brings up the Advanced Distribution Kit Options dialog box.

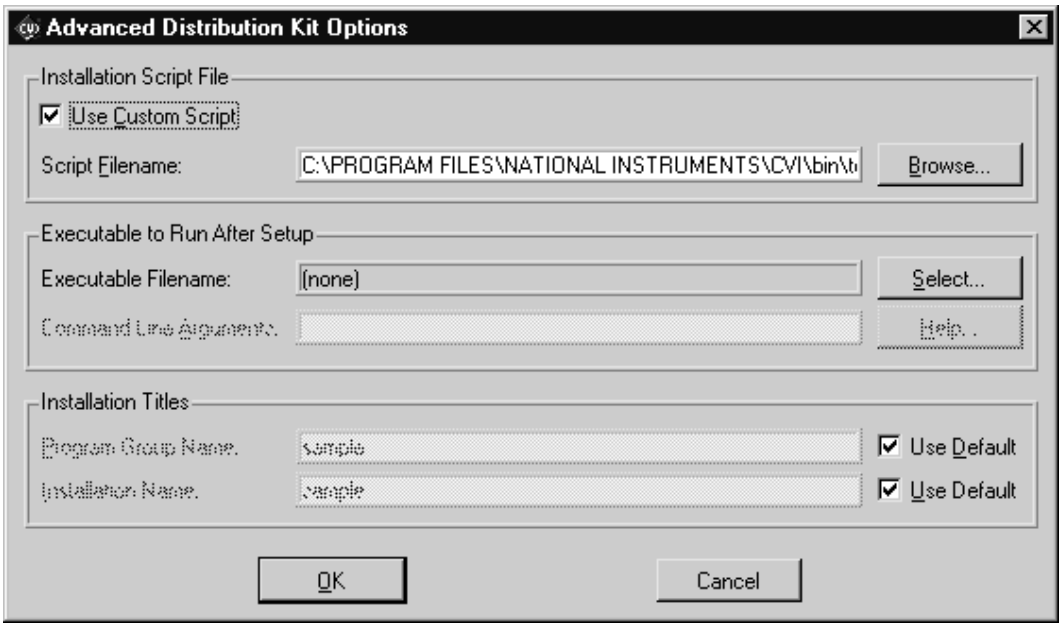


Figure 2-4. Advanced Distribution Kit Options dialog box

Installation Script File Section

- **Use Custom Script**—When this checkbox is selected, you can enter the name of a customized installation script file for your distribution kit. The default installation script file is `cvi\bin\template.inf`.
- **Script Filename**—The pathname of the customized installation script file. You can use the **Browse** button to select an existing filename.

Executable to Run After Setup

- **Executable Filename**—The name of an executable file to run after the end-user installation is complete. Use the **Select** button to select a file that has already been added to one of the file groups.
- **Command Line Arguments**—The command line arguments to pass to the executable run after the installation is complete. Use the Help button to view detailed information on special macros you can use in this control.

Installation Titles

- **Program Group Name**—The name of the program group created during the installation. If the **Use Default** button is checked, the following priority is used to determine the program group name.
  - If the project target is an executable, the application title entered in the Create Standalone Executable dialog box is used, if it is non-empty.
  - Otherwise, the base file name of the target file (executable, DLL, or static library) is used, if it has been created.
  - Otherwise, the base file name of the project is used.
- **Installation Name** -- The installation window title and the text displayed in upper part of the installation window. If the **Use Default** button is checked, the name is set using the same priority as for the **Program Group Name**.

Using Instrument Drivers

This section discusses the changes that have been made with regard to using instrument drivers.

Instrument Driver Files

In general, all instrument driver files must be in the same directory. However, some exceptions have been introduced.

VXIplug&play Include Files

When you install a VXI*plug&play* instrument driver, the include (`.h`) file is placed in a different directory than the directory of the function panel (`.fp`) file. LabWindows/CVI can find include files in the VXI*plug&play* include directory.

(For more information, see the *Include Paths* section in Chapter 1, *Updates to the Programmer Reference Manual* in this document.)

VXIplug&play DLLs (Windows 3.1)

When you install a VXI*plug&play* instrument driver, its DLL is placed in a different directory than the function panel (`.fp`) file. The directory containing the DLL is listed in the PATH environment variable. If LabWindows/CVI cannot find a program file for the instrument driver in the same directory as the `.fp` file, it searches for the DLL in using the standard Windows DLL search algorithm, which includes all directories listed in the PATH environment variable.

(For more information, see the *Searching for Instrument Driver DLLs (Windows 3.1 Only)* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document.)

DLL Import Libraries for VXI Plug & Play DLLs (Windows 95 and NT)

In LabWindows/CVI for Windows 95 and NT, DLLs must always be accompanied by a DLL import library (`.lib`) file. The `.lib` file is considered to be the program file for the instrument driver. LabWindows/CVI never looks for a DLL file directly, and cannot use DLL path (`.pth`) files.

Generally, the DLL import library must be in the directory of the function panel (`.fp`) file. However, when you install a VXI*plug&play* instrument driver DLL, its DLL import library is placed in a different directory than the `.fp` file. The import libraries are placed in subdirectories in the `LIB` directory under the VXI*plug&play* framework directory. The import libraries that work with Visual C/C++ are placed in the `MSC` subdirectory. The import libraries that work with Borland C/C++ are placed in the `BC` subdirectory. (Import libraries for WATCOM C/C++ or Symantec are not distributed.)

If the `.fp` file is under the VXI*plug&play* directory tree, LabWindows/CVI looks for a `.lib` file in the VXI*plug&play* import library subdirectory for the current compatible compiler. (For WATCOM C/C++, it looks for a WC subdirectory. For Symantec C/C++, it looks for an SC directory.) LabWindows/CVI looks for the VXI*plug&play* import library before it looks for a program file in the directory of the `.fp` file, unless a program file in the directory of the `.fp` file (and with the same base name as the `.fp` file) is listed in the project and is unexcluded.

(For more information, see the *Loading 32-bit DLLs under Windows 95 and NT* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document.)

Window Menu

The following commands have been added to the **Window** menu under the Tile Windows command.

Minimize All (Windows 95 only)

The Minimize All command hides all of the LabWindows/CVI windows, including the Project window and any User Interface Library panels displayed by a program you are running in LabWindows/CVI. You can restore the windows by clicking on LabWindows/CVI in the Windows 95 task bar.

CloseAll

The **Close All** command closes all of the LabWindows/CVI windows, excluding the Project window and any User Interface Library panels displayed by a program you are running in LabWindows/CVI. It works on all platforms.

Library Menu

This section discusses the changes to the **Library** menu in the Project window and all other windows in which the **Library** menu appears.

Easy I/O for DAQ (Windows 3.1, Windows 95 and NT)

The Easy I/O for DAQ library has been added to the **Library** menu.

In LabWindows/CVI version 3.1, Easy I/O for DAQ was distributed as an instrument driver. If you included `easyio.fp` in a project created in LabWindows/CVI version 3.1, you should remove it when you bring the project into LabWindows/CVI version 4.0.

For detailed descriptions of the enhancements made to the Easy I/O for DAQ library see Chapter 4, *Updates to the Standard Libraries Reference Manual*, in this document.

Options Menu

This section discusses the changes to the **Options** menu in the Project window.

Compiler Options

For Windows 95 and NT, the compatible compiler is indicated in the **Compiler Options** dialog box. (For more information on external compiler compatibility, see the *Compatibility with External Compilers* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document.)

For Windows 95 and NT, the default calling convention can be changed in the **Compiler Options** dialog box, unless the compatible compiler is WATCOM. For the other compilers, the default calling convention is normally `cdecl` but can be changed to `stdcall`. For WATCOM, it is always the stack-based calling convention. (For more information, see the *Calling Conventions (Windows 95/NT Only)* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document.)

Compiler Defines

For Windows 95 and NT, the default compiler defines string now contains

```
/DWIN32_MEAN_AND_LEAN
```

This is to reduce the time and memory taken by compiling Windows SDK include files. (For more information, see the *Calling Windows SDK Functions in LabWindows/CVI* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document, in this document.)

For all platforms, the **Compiler Defines** dialog box now contains a list of the macros predefined by LabWindows/CVI. This list includes the name and value of each predefined macro.

Include Paths

The **Include Paths** dialog box has been modified so that there are now two lists of paths in which to search for include files. The top list is saved with the project file. The bottom list is saved from one LabWindows/CVI session to another on the same machine, regardless of project. (For more information, see the *Include Paths* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document.)

Run Options

For Windows 95 and NT, the option **Reload DLLs After Each Run** has been renamed to **Unload DLLs After Each Run**. Also, the option is now enabled by default. (For more information, see the *Default Unloading/Reloading Policy* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document.)

Source Window Changes

This chapter discusses the changes to the Source window, including the menu commands accessible from the Source window. This chapter generally follows the organization of Chapter 3, *Source, Interactive Execution, and Standard Input/Output Window*, of the *User Manual*. The changes apply to all platforms, unless otherwise marked.

Notification of External Modification (Windows 3.1 and Windows 95/NT Only)

If a file in a source window has been modified externally because it was last loaded or saved by LabWindows/CVI, then when you switch back to LabWindows/CVI from another Windows application, a dialog box appears. You are given the option of updating the source window from the file on disk, overwriting the file on disk with the contents of the source window, or doing nothing.

Backspace to Beginning of Word

The **Backspace to Beginning of Word** command is available only from the keyboard. It removes all text from the current position in the Source window to the beginning of the previous word in the file. You invoke the command by pressing <Ctrl-Shift-Backspace>.

Context Menus

You can bring up a context menu in the Source window by pressing the right mouse button. The context menu contains a selection from the most commonly used menu commands from the Source window menu bar. The selection of commands is different depending on whether the mouse is over the text editing area or over the line number or line icon area.

Edit Menu

The following command has been added to the **Edit** menu.

Select All

The **Select All** command selects all of the text in the source window, and positions the keyboard cursor at the end of the file.

View Menu

The following enhancements have been made to commands in the **View** menu.

Recall Panel

Improvements have been made in the algorithm which recognizes the function name in the source code text.

- If you place the keyboard cursor over a function name, **Recall Panel** recognizes the function name even if it is not followed by a parameter list. Thus, you can simply type a function name into the Source window and execute **Recall Panel**.
- **Recall Panel** can find a function call within an expression. For example, if you place the keyboard cursor at the beginning of the following line

  ```
  if ((x = fn (a, b)) == 0)
  ```

  Recall Panel finds the function call

  ```
  x = fn (a, b)
  ```

- If a function call is split among several lines, you can place the keyboard cursor on any of the lines and **Recall Panel** can still find the function call.

Find Function Panel

In previous versions of LabWindows/CVI, the **Find Function Panel** command required that you enter the complete name of a function. In effect, it was only matching on the *whole word*.

Now, an explicit **Whole Word** option has been added, and it is *disabled by default*. Thus, you can enter just a substring, and Find Function Panel finds all functions that contain that substring anywhere in their names. For instance, if you enter

```
ctrl
```

and click on **OK**, a dialog box appears with a list of functions including `NewCtrl`, `SetCtrlVal`, `GetCtrlVal`, and so on.

Also, a shortcut key, <Ctrl-Shift-P> has been added for the **Find Function Panel** command.

Run Menu

The following changes and additions have been made to the **Run** menu.

Terminate Execution Shortcut Key Changed for Windows 95/NT

The short-cut key for terminating execution of a suspended program or suspending a running program is <Ctrl>-F12 on Windows 95 and NT.

Activate Panels When Resuming

You can use **Activate Panels when Resuming** to choose whether the user interface panels in your programs are reactivated every time you resume execution during debugging. By default, this option is enabled. Activating the panels whenever you resume guarantees that the activation state of every panel is identical to what it would be if you were not debugging. In general, however, this is not important, and activating panels each time you resume can be time consuming.

If you disable this option, your panels are activated when your program causes events to be processed or explicitly displays, activates, hides, or discards panels.

This option is saved from CVI session to session, not in the project file.

Options Menu

The following commands have been added to the **Options** menu or modified for Windows 95/NT.

Colors

The Color dialog box contains eight new color types for syntax coloring. Refer to the *Syntax Coloring* section later in this chapter for more details.

A new checkbox, **Use System Colors**, has been added for Windows 95. When enabled, this option removes from the list box several color types associated with the Project window, Source

window, and scroll bars. Colors are automatically assigned to these types based on the system colors defined in the Appearance tab in the Windows 95 Display Properties dialog box.

Syntax Coloring

When you enable the **Syntax Coloring** option, LabWindows/CVI color codes the various types of tokens in your source and include files. The following are the different types of tokens that can be color coded.

- C keywords
- identifiers
- comments
- integers
- real numbers
- strings
- preprocessor directives
- user-defined tokens

You can set the color for token type via the **Color** command in the **Options** menu.

You can create the list of user-defined tokens via the **User Defined Tokens for Coloring** command in the **Options** menu.

User Defined Tokens for Coloring

You can use the **User Defined Tokens for Coloring** command to define tokens that can be displayed in a unique color when the **Syntax Coloring** option is enabled. You use the **Colors** command to set the color. Each token must be in the form of a valid C identifier. You can cause a token to be saved in your project file or saved from one CVI session to another without regard to which project is loaded.

Generate DLL Import Source (Windows 95/NT Only)

(This command replaces the **Generate DLL Glue Source** command available in LabWindows/CVI for Windows 3.1.)

This command generates source code that can be used to create a DLL import library. In general, you do not need to use this command. For most cases, you can generate a DLL import library directly using the **Generate DLL Import Library** command. Use this command only when you

must do special processing in the DLL import library. LabWindows/CVI never requires such special processing.

The **Generate DLL Import Source** command is enabled only when you have an include file in the Source window. The include file should contain declarations of all of the functions you want to access from the DLL. When you execute the command a file dialog box appears. Enter the pathname of the DLL.

The command generates the import library source into a new Source window. You can modify the code, including making calls to functions in other source files. Create a new project containing the source file and any other files it references. Select **Static Library** from the submenu attached to the **Target** command in the **Build** menu of the Project window. Execute the **Create Static Library** command.

Note: ***You cannot export variables from a DLL using the import library source code generated by this command. When you want to export a variable, create functions to get and set its value or create a function to return a pointer to the variable.***

Note: ***When you edit the source code generated by this command, you cannot use the*** `__import` ***qualifier on the function declarations in the DLL include file.***

Note: ***The import source code does not operate in the same way as a normal DLL import library. When you link a normal DLL import library into an executable, the operating system attempts to load the DLL as soon as the program starts. The import source generated by LabWindows/CVI is written so that the DLL is not loaded until the first function call into it is made.***

Generate DLL Import Library (Windows 95/NT Only)

(This command replaces the **Generate DLL Glue Code** command available in LabWindows/CVI for Windows 3.1.)

This command generates a DLL import library. The command is enabled only when you have an include file in the Source window. The include file should contain declarations of all of the functions and global variables you want to access from the DLL. When you execute the command, you have the option to generate an import library for each of the compatible external compilers rather than just for the current compatible compiler. A file dialog box then appears. Enter the pathname of the DLL.

The command generates a `.lib` file with the same base name as the include file. If you choose to create an import library for each compiler, the files are created in subdirectories named `MSVC`, `BORLAND`, `WATCOM`, and `SYMANTEC`. The library for the current compatible compiler is also created in the directory of the DLL.

Create Object File

For Windows 95 and NT, the **Create Object File** command has been modified so that it gives you the option of creating an object file for each of the compatible external compilers rather than just for the current compatible compiler. If you chose to create an object file for each compiler, the files are created in subdirectories named `MSVC`, `BORLAND`, `WATCOM`, and `SYMANTEC`. The object file for the current compatible compiler is also created in the parent directory.

Function Panel Changes

This chapter discusses the changes to the Function Panel window, including the menu commands accessible from the Function Panel window. This chapter generally follows the organization of Chapter 4, U*sing Function Panels*, of the *User Manual*. The changes apply to all platforms, unless otherwise marked.

Code Menu

The following command has been added to the **Code** menu.

Select Variable

The **Select Variable** command gives you a list of previously used variables or expressions having data types that are compatible with the currently active function panel control. The command is enabled only when the currently active function panel control is one that accepts text entry. When you select a variable or expression from the list, it is copied into the function panel control. The Select Variable command can significantly reduce the amount of keyboard entry needed when using function panels.

When you execute the **Select Variable** command, the Select Variable or Expression dialog box appears.

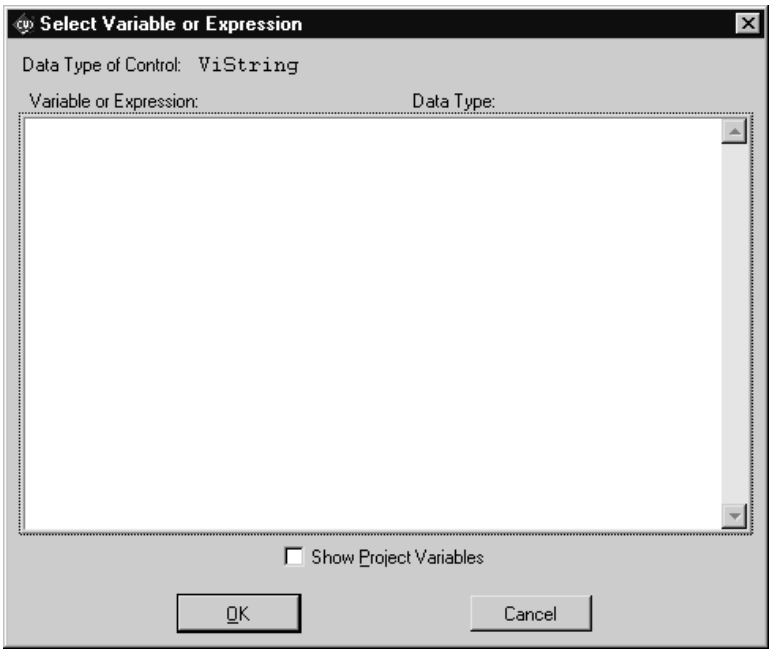


Figure 2-5. The Select Variable or Expression Dialog Box

- **Data Type of Control**—Indicates the data type of the currently active function panel control.
- **Variable or Expression**—This list box column contains the variables and expressions that have data types compatible with the data type of the control.
- **Data Type**—This list box column indicates the data type of each variable and expression.
- **Show Project Variables**—This option adds to the list box global variables (both static and non-static) defined in project files that have been successfully compiled.
- **OK**—This button dismisses the dialog box and copies the variable or expression into the function panel control. It may add a leading ampersand (&) when the function panel control is an output control. It may add one or more leading asterisks (*) or a trailing array indexation (`[0]`) when needed to correctly match the data type of the control.
- **Cancel**—This button cancels the operation.

What Can be Included in the List Box

The following items are considered for inclusion in the list box.

- Variables declared in the Interactive Execution window.
- Variables declared using the **Declare Variable** command in a function panel.

- Variables or expressions used in function panels that are executed.
- Variables or expressions used the function panels from which code is inserted into a source window.
- User interface panel handle variables added to a source window by CodeBuilder.
- Variables declared as global or static global in a project file that has been successfully compiled, but only if the **Show Project Variables** option has been enabled in the dialog box.

Some or all of these items are cleared from memory when you unload the current project or execute the **Clear Interactive Declarations** command in the **Build** menu.

Data Type Compatibility

Compatibility between data types is a more complex issue than might be expected. In the end, heuristics must be used. The heuristics differ based on whether the variable is known to the compiler.

Variables known to the compiler include variables declared in the Interactive Window, and variables declared in project files that have been successfully compiled. For such variables, the following are the major factors in determining whether the variable is type-compatible with a function panel control.

- Data types declared with the `typedef` keyword are reduced to their most intrinsic type, as long as the `typedef` is known to the compiler. For example, assume the following declarations have been processed by the compiler.

  ```
  typedef int   typeA;
  typedef int   typeB;
  typedef typeB typeC;
  ```

  Then a variable of type `typeA` is an exact match for a function panel control having type `typeC`.

- All numeric types are considered compatible with each other, except that floating point variables or expressions are not considered compatible with integer function panel controls.
- Types that have the same base type but differ in levels of indirection are considered to be compatible. For example, the following are all compatible:

  ```
  int
  int *
  int **
  int [];
  ```

An expression or a variable name not known to the compiler must match exactly to the function panel control's data type to be included in the list box. (An example of a variable name not

known to the compiler is one used in a function panel from which code has been inserted into a source window.)

Note: ***An expression or variable name not known to the compiler can be associated with multiple data types. For instance, you might use the same variable name in an*** `int` ***control and a*** `double` ***control. If the variable is not known to the compiler, LabWindows/CVI has no way of knowing the true data type of the variable name. Thus, you might see the variable name associated with different data types.***

Sorting of List Box Entries

The entries in the list box are first sorted by data type. The most compatible data types are shown first. (Exception: Some function panel controls are declared with "meta" data types, such as `numeric array`, `any array`, or `any type`. Such controls are equally compatible with a wide range of data types. In this case, the order of data types does not indicate differing degrees of compatibility.)

Within each data type, the entries are sorted alphabetically by the variable/expression text.

Chapter 3
Updates to the User Interface Reference Manual

Chapter Contents

Changes to the User Interface Library

This section presents changes made to the LabWindows/CVI User Interface Library.

The User Interface library is documented in the *User Interface Reference Manual*.

Summary of Major Enhancements

The major enhancements made the User Interface library are the following.

- A new control type, *canvas*, has been added. You can execute arbitrary drawing commands within a canvas control. A set of canvas drawing functions have been added. Functions have also been added for handling rectangles, points, and bitmaps.
- Functions have been added that you can use to copy text and images to and from the Windows system clipboard.
- You can now have a second Y axis on a graph.
- You can now scale the values shown on graph and strip chart axes.
- You can now reverse the polarity of the graph and strip chart axes.
- You can now create your own strings for tick labels on graphs and strip charts.
- You can now use interactive zooming and panning on graphs.
- You can now use 24-bit pixel depths with bitmap images.
- On Windows 95, the native system dialog boxes are used for the `FileSelectPopup`, `MultiFileSelectPopup`, and `DirSelectPopup` functions.
- You can now use Windows metafiles (`.WMF`) as an image type.

Corrections to Documentation

The following corrections should be made to the *LabWindows/CVI User Interface Reference Manual*.

VAL_PORTRAIT and VAL_LANDSCAPE Values

In Table 3-21 on page 3-74 of the *LabWindows/CVI User Interface Reference Manual*, the numeric values shown for `VAL_PORTRAIT` and `VAL_LANDSCAPE` are incorrect. The following are the correct values.

```
1 = VAL_PORTRAIT

2 = VAL_LANDSCAPE
```

RegisterWinMsgCallback

In the documentation for the `RegisterWinMsgCallback` function on page 4-132 of the *LabWindows/CVI User Interface Reference Manual*, the description for the **messageIdentifier** parameter should be changed to the following.

A user-defined string that allows two processes to use the same Windows message number. If you pass zero (0), a unique Windows message number is generated. If you pass a string, all subsequent calls to `RegisterWinMsgCallback` (or `RegisterWindowMessage` in the Windows API) using the same **messageIdentifier** string will return the same message number.

Using Zooming and Panning on Graph Controls

The following should be added to the *Graph Controls* section in Chapter 1, *User Interface Concepts* in the *LabWindows/CVI User Interface Reference Manual*.

Zooming and Panning on Graphs

You can use *zooming*—the ability expand or contract the viewport around a particular point—in graph controls. When you zoom *in*, the logical area contained in the viewport gets smaller, thereby showing the area with more resolution. When you zoom *out*, the viewport shows a wider area. You can also use *panning*, the ability to shift the viewport.

By default, however, zooming and panning are disabled. You must explicitly enable them in the User Interface Editor or programmatically. Also, a graph control must not be in indicator-only mode if zooming and panning are to be used.

To start zooming in on a point, press the <Ctrl> key and left mouse button down over the point. The resolution in the viewport is continuously increased until you release the mouse. (You do not need to keep the <Ctrl> key down.) If you drag the mouse, the zooming continues but does so over the new point under the mouse cursor. The zooming stops when you release the left mouse button or click on the right mouse button.

You zoom out just like you zoom in, except that you use the right mouse button instead of the left mouse button.

To start panning, press the <Ctrl-Shift> keys and the left mouse button over a point on the viewport. Then drag the mouse to another point. The graph viewport is scrolled so that the original point now appears under the new mouse cursor location. You can drag the mouse anywhere on the screen.

To restore the viewport to its original state after zooming or panning, press <Ctrl-Spacebar>.

If you are using auto-scaling in the graph, the auto-scaling must be temporarily disabled while zooming or panning. If any plotting occurs while the end-user is zooming or panning, the zooming or panning is terminated and the new data is shown using auto-scaling.

Using Canvas Controls

The following section should be added after the *Timer Controls* section in Chapter 1, *User Interface Concepts* in the *LabWindows/CVI User Interface Reference Manual*.

Canvas Controls

Use *canvas controls* as an arbitrary drawing surface. You can draw text, shapes, and bitmap images. An offscreen bitmap is maintained so that the appearance of the canvas can be restored when the region is exposed.

If you want to display images that are not rectangular or that have “holes” in them, you can use bitmaps that have a transparent background.

CodeBuilder Changes

The following changes have been made to the **Generate** » **All Code** and **Generate » Main Function** commands in the **Code** menu of the User Interface Editor window. (These commands are discussed in Chapter 2, *User Interface Editor Reference*, of the *LabWindows/CVI User Interface Reference Manual*).

WinMain

A new checkbox, **Generate WinMain() instead of main()**, has been added to the Generate All Code and Generate Main Function dialog boxes. Enable this option if you want to use `WinMain` instead of `main` for your main program. In LabWindows/CVI, you can use either function as your main program. When linking your application in an external compiler, it is easier to use `WinMain`.

DLL Projects

If your project target is a DLL, neither `WinMain` or `main` are generated. Instead, a `DLLMain` function is generated. The bulk of the User Interface function calls, however, are generated in a

function call `InitUIForDLL`. You can call `InitUIForDLL` in your DLL at the point you want to load and display panels.

InitCVIRTE and CloseCVIRTE Functions

When you link your executable or DLL in an external compiler, you need to include a call to the `InitCVIRTE` function in `WinMain`, `main`, or `DLLMain`. In a DLL, you also need to include a call to `CloseCVIRTE`. See the *Calling `InitCVIRTE` and `CloseCVIRTE`* section in Chapter 1, *Updates to the Programmer Reference Manual*, in this document.

CodeBuilder automatically generates the necessary calls to `InitCVIRTE` and `CloseCVIRTE` in your `WinMain`, `main`, or `DLLMain` function. It also automatically generates a `#include` statement for the `cvirte.h` file. It generates this code within comment markers. If you are going to use an external compiler, remove the comment markers from these lines.

New Qualifier for Callback Functions

The following paragraph should be added before the last paragraph in the *Using Callback Functions to Respond to User Interface Events* section in Chapter 3 of the *User Interface Reference Manual*.

The `CVICALLBACK` macro should precede the function name in the declarations and function headers for all user interface callbacks. This ensures that the functions are treated by the compiler as `cdecl` (or stack-based in WATCOM), even when the default calling convention is `stdcall`. `CVICALLBACK` is defined in `cvidefs.h`, which is included by `userint.h`. The `CVICALLBACK` macro is included where necessary in the header files generated by the User Interface Editor and in source code generated by CodeBuilder.

Additions to Table 3-2, Panel Attributes

The following new information belongs in Table 3-2 of the *LabWindows/CVI User Interface Reference Manual*.

`ATTR_ACTIVE`	`int`	Indicates whether the panel or one of its child panels is the active panel. (`GetPanelAttribute` only.)
`ATTR_CONFORM_TO_SYSTEM`	`int`	Specifies whether the panel and its controls use the system colors. Subsequent new controls also use system colors. (This is useful only on Windows 95. Other platforms always use panel gray and black .)
`ATTR_FLOATING`	`int`	Specifies whether the panel floats above all non-floating panels. Applies to top-level panels in Microsoft Windows.

Additions to Table 3-5, Font Values

The following new information belongs in Table 3-5 of the *LabWindows/CVI User Interface Reference Manual*.

Type	Value
Platform independent font	VAL_MESSAGE_BOX_FONT
Platform independent metafont	VAL_MESSAGE_BOX_META_FONT

Also, add the following to the *Platform-Independent Metafonts That Are Resident on PCs and UNIX* section on page 3-19 of the *LabWindows/CVI User Interface Reference Manual*.

VAL_MESSAGE_BOX_META_FONT is the font used by LabWindows/CVI for message boxes. On Windows 95, it is the font used by Windows 95 for message boxes. On other platforms, it is the same as VAL_DIALOG_META_FONT.

Additions to Table 3-6, Menu and Menu Item Attributes

The following new information belongs in Table 3-6 of the *LabWindows/CVI User Interface Reference Manual*.

Menu Bar Only	ATTR_DRAW_LIGHT_BEVEL	int	Indicates whether the menubar draws with a light or dark bevel at the bottom. (Applies only to Windows 95.) 0 = Dark Bevel (the default) 1 = Light Bevel

Additions to Table 3-7, Key Modifiers and Virtual Keys

The following new information belongs in Table 3-7 of the *LabWindows/CVI User Interface Reference Manual*.

Key Modifiers	VAL_UNDERLINE_MODIFIER <ALT> on the PC and the SPARCstation. Also, optionally, <META> on the SPARCstation.

Additions to Table 3-9, Control Attributes

The following new information belongs in Table 3-9 of the *LabWindows/CVI User Interface Reference Manual*.

For all controls

ATTR_OVERLAPPED	int	Indicates whether the control is overlapped by another control (or its own parts). (GetCtrlAttribute only)

For list boxes

ATTR_ALLOW_ROOM_FOR_IMAGES	int	Specifies whether, when calculating the list box height, to assume that one or more list box labels may contain an image. Normally, the calculation of the list box height takes into account the image height only if there currently is an image in the list box. To make sure that the list box height always takes into account the height of an image, set this attribute to TRUE (1).
ATTR_HILITE_CURRENT_ITEM	int	Specifies whether to highlight the currently selected item in a list box. (The highlight is shown in reversed colors when list box is active, a dashed box when inactive.)

ATTR_PICT_BGCOLOR now applies to canvas controls, as well as to picture controls and picture rings.

Addition to Table 3-10, Control Styles for ATTR_CTRL_STYLE

The following new information belongs in Table 3-10 of the *LabWindows/CVI User Interface Reference Manual* .

Canvas	CTRL_CANVAS	

Programming with Canvas Controls

Use a canvas control to add an arbitrary drawing surface to your project. You can draw text, shapes, and bitmap images. This section describes how you can use the User Interface Library functions and attributes with canvas controls.

Functions for Drawing on Canvas

Use the following functions to draw on a canvas.

- `CanvasDrawPoint` to draw a point.
- `CanvasDrawLine` to draw a line.
- `CanvasDrawLineTo` to draw a line from the current pen position.
- `CanvasDrawRect` to draw a rectangle.
- `CanvasDimRect` to overlay a checkerboard pattern in a rectangular area.
- `CanvasDrawRoundedRect` to draw a rectangle with rounded corners.
- `CanvasDrawOval` to draw an oval.
- `CanvasDrawArc` to draw an arc.
- `CanvasDrawPoly` to draw a polygon.
- `CanvasDrawText` to draw text within a rectangular area.
- `CanvasDrawTextAtAPoint` to draw text at an anchor point.
- `CanvasDrawBitmap` to draw a bitmap image.
- `CanvasScroll` to scroll a rectangular area.
- `CanvasInvertRect` to invert the colors in a rectangular area.
- `CanvasClear` to restore a rectangular area to the canvas background color.

Batch Drawing

Although, the drawing functions can be called at any time, they are most efficient when called from within a batch drawing operation. A batch drawing operation consists of a call to `CanvasStartBatchDraw`, followed by one or more calls to the canvas drawing functions, followed by a call to `CanvasEndBatchDraw`.

For optimal performance, users are encouraged to include as many drawing primitives as possible within a batch drawing operation. When a drawing function is called outside of a batch operation, the function is implicitly surrounded by calls to `CanvasStartBatchDraw` and `CanvasEndBatchDraw`.

Canvas Coordinate System

A canvas has a built-in pixel-based Cartesian coordinate system, where (0,0) represents the top, left corner of the canvas. All drawing is specified relative to this coordinate system. The coordinate system can be modified using the following four attributes:

```
ATTR_CANVAS_XCOORD_AT_ORIGIN
ATTR_CANVAS_YCOORD_AT_ORIGIN
ATTR_CANVAS_XSCALING
ATTR_CANVAS_YSCALING
```

All canvas control functions use this coordinate system, except for `CanvasGetPixel` and `CanvasGetPixels`, which use unscaled pixel coordinates rather than the canvas coordinate system.

Offscreen Bitmap

Each canvas has an offscreen bitmap which is used to restore the appearance of the canvas when the region is exposed. You can choose to draw directly to the screen, bypassing the offscreen bitmap. If you draw to the offscreen bitmap, you can choose whether to update the screen immediately or wait until draw events are processed. This is controlled by the `ATTR_DRAW_POLICY` attribute.

The `CanvasUpdate` function immediately copies the canvas offscreen bitmap to the screen, within a specified rectangular area.

The `ATTR_OVERLAP_POLICY` attribute controls what occurs when you draw to a canvas which is overlapped by another control.

Clipping

The drawing functions are constrained by the clipping set using `CanvasSetClipRect`. Any drawing outside the clipping rectangle is not rendered. You can obtain the current clipping rectangle by calling `CanvasSetClipRect.`

Background Color

The background color of the canvas is controlled by the `ATTR_PICT_BGCOLOR` attribute. When `ATTR_PICT_BGCOLOR` is changed, the entire canvas area is cleared.

Pens

Each canvas has a pen. The canvas pen attributes can be set individually using `SetCtrlAttribute`. They are:

```
ATTR_PEN_WIDTH
ATTR_PEN_STYLE
ATTR_PEN_COLOR
ATTR_PEN_FILL_COLOR
ATTR_PEN_MODE
ATTR_PEN_PATTERN
```

The `CanvasDefaultPen` function resets all these attributes to their default values.

The location of the pen affects the starting position of the line drawn by the `CanvasDrawLineTo` function. The location of the pen is affected only by the `CanvasSetPenPosition` and the `CanvasDrawLineTo` functions. You can obtain the location of the pen by calling `CanvasGetPenPosition`.

Pixel Values

You can obtain the color values of pixels in the canvas. Call `CanvasGetPixel` to obtain the color of one pixel. Call `CanvasGetPixels` to obtain the values of the pixels within a rectangular area. The color values are obtained from the offscreen bitmap, not the screen.

Unlike other canvas control functions, `CanvasGetPixel` and `CanvasGetPixels` use unscaled pixel coordinates rather than the canvas coordinate system.

Canvas Attributes

The following table lists attributes unique to the canvas control. You can access these attributes through `GetCtrlAttribute` and `SetCtrlAttribute`.

Table 3-1. Canvas Control Attributes

For canvas controls		
Name	**Type**	**Description**
ATTR_DRAW_POLICY	int	Determines when drawing operations are rendered on the offscreen bitmap and the screen. See discussion that follows this table.
ATTR_OVERLAPPED_POLICY	int	Determines what occurs when you draw to a canvas which is overlapped by another control. See discussion that follows this table.
ATTR_PEN_COLOR	int	The RGB color value used to draw points, lines, frames, and text on the canvas.
ATTR_PEN_FILL_COLOR	int	The RGB color value used to fill interior areas of shapes, text backgrounds, and areas exposed by scrolling.
ATTR_PEN_MODE	int	Determines the effect of drawing with the pen color (or pen fill color), given the current color on the screen. See discussion following this table.
ATTR_PEN_PATTERN	unsigned char[8]	Determines the pattern used to fill interior areas of shapes. See discussion following this table.
ATTR_PEN_STYLE	int	The style used when drawing lines and frames. In Windows, applies only if pen width is 1; if pen width is greater than 1, the style is always VAL_SOLID. See Table 3-19 in the *LabWindows/CVI User Interface Reference Manual*.
ATTR_PEN_WIDTH	int	The number of pixels in the width of a pen stroke. Applies to lines, frames, and points. Valid range: 1 to 255. Default: 1.
ATTR_XCOORD_AT_ORIGIN	double	The horizontal coordinate mapped to the left edge of the canvas. (Is multiplied by ATTR_XSCALING to arrive at a pixel offset.) Default value: 0.0.
ATTR_XSCALING	double	The factor used to scale user-supplied horizontal coordinates and widths into pixel-based coordinates and widths. Default value: 1.0.
ATTR_YCOORD_AT_ORIGIN	double	The vertical coordinate mapped to the top edge of the canvas. (Is multiplied by ATTR_YSCALING to arrive at a pixel offset.) Default value: 0.0.
ATTR_YSCALING	double	The factor used to scale user-supplied vertical coordinates and heights into pixel-based coordinates and heights. Default value: 1.0.

Canvas Attribute Discussion

The following table lists the values associated with the ATTR_DRAW_POLICY attribute.

Table 3-2. Values for ATTR_DRAW_POLICY

Value	Description
VAL_UPDATE_IMMEDIATELY	Drawing takes place offscreen. The section of the bitmap corresponding to the area of the drawing operation is copied into the canvas display immediately. (This is the default.)
VAL_MARK_FOR_UPDATE	Drawing takes place offscreen. The area on the canvas corresponding to the area of the drawing operation is marked for update. The new drawing becomes visible when draw events are processed.
VAL_DIRECT_TO_SCREEN	Drawing goes directly to the screen. The offscreen bitmap is not updated. Although this may result in a faster drawing time, whatever is drawn in this mode is lost when the canvas is redrawn.

The following table lists the values associated with the ATTR_OVERLAPPED_POLICY attribute.

Table 3-3. Values for ATTR_OVERLAPPED_POLICY

Value	Description
VAL_DEFER_DRAWING	If the control is overlapped and the draw policy is VAL_UPDATE_IMMEDIATELY, new drawing does not become visible until draw events are processed. If the draw policy is VAL_DIRECT_TO_SCREEN, no drawing takes place at all. (This is the default.)
VAL_DRAW_ON_TOP	If the control is overlapped and the draw policy is *not* VAL_MARK_FOR_UPDATE, drawing occurs on top of the overlapping controls.

The ATTR_PEN_MODE attribute determines the effect of drawing with the pen color (or pen fill color), given the current color on the screen. With the default setting, VAL_COPY_MODE, the current screen color is replaced with the pen color (or pen fill color). The other settings specify bitwise logical operations on pen color (or pen fill color) and the screen color.

Note: ***If a system color palette is in use, the logical operations might be performed on the palette indices rather than the RGB values, depending on the operating system.***

The following table lists the values associated with the ATTR_PEN_MODE attribute.

Table 3-4. Values for ATTR_PEN_MODE

VAL_COPY_MODE	pen color (the default)
VAL_OR_MODE	pen color | screen color
VAL_XOR_MODE	pen color ^ screen color
VAL_AND_NOT_MODE	~(pen color) & screen color
VAL_NOT_COPY_MODE	~(pen color)
VAL_OR_NOT_MODE	~(pen color) | screen color
VAL_NOT_XOR_MODE	~(pen color ^ screen color)
VAL_AND_MODE	pen color & screen color

The ATTR_PEN_PATTERN attribute determines the pattern used to fill interior areas of shapes. The value is an 8-byte unsigned character array representing a repeating 8-by-8 grid of pixels through which filling operations are filtered. A pixel of value 1 means that the pen fill color is used for that pixel. A pixel value of 0 means that black is used for that pixel. The default value for the attribute is the solid pattern, in which each byte of the array is 0xFF.

To make a pixel value of 0 mean "screen color" instead of "black", do the following.

1. Set ATTR_PEN_PATTERN to the complement of the pattern you wish to use.
2. Set ATTR_PEN_MODE to VAL_AND_MODE.
3. Set ATTR_PEN_FILL_COLOR to VAL_WHITE.
4. Use a canvas draw function, (for example, CanvasDrawRect) to fill the area.
5. Set ATTR_PEN_PATTERN to the desired pattern.
6. Change the ATTR_PEN_MODE to VAL_OR_MODE.
7. Change the ATTR_PEN_FILL_COLOR to the desired pattern color.
8. Draw again.

Using Rect and Point Structures

Two structures, Rect and Point are defined in the userint.h include file. These structures are used to specify locations and areas in Cartesian coordinate systems, such as those used in canvas controls and bitmaps. Many canvas control functions use these structures.

The Rect structure specifies the location and size of a rectangle. It is defined as follows.

```
typedef struct
  {
  int top;
  int left;
  int height;
  int width;
  } Rect;
```

A `Point` structure specifies the location of a point. It is defined as follows.

```
typedef struct
  {
  int x;
  int y;
  } Point;
```

Functions and Macros for Making Rects and Points

You might need to create a `Rect` or `Point` just to pass it to a function. You can avoid creating a variable for this by using one of the following functions.

`Rect` **MakeRect** (`int` **top**, `int` **left**, `int` **height**, `int` **width**);
`Point` **MakePoint** (`int` **x**, `int` **y**);

For example,

```
CanvasDrawPoint (panel, ctrl, MakePoint (30, 40));
```

You can also use these function to initialize variables. For example,

```
Rect r = MakeRect (10, 20, 100, 130);
```

There are special values for the `Rect` height and width. Also, there are some macros for creating commonly used rectangles. The documentation for each function indicates when these values and macros are applicable. See the following table.

Table 3-5. Values and Macros for Rect Structures

Name	Value or Definition	Description
`VAL_TO_EDGE`	`-1`	Set the `Rect` width (or height) to the distance from the `Rect` left (or top) to the right (or bottom) edge of the object.
`VAL_KEEP_SAME_SIZE`	`-2`	When copying objects (such as bitmaps), make the destination object the same size as the source object.
`VAL_EMPTY_RECT`	`MakeRect (0, 0, 0, 0)`	An empty rectangle.
`VAL_ENTIRE_OBJECT`	`MakeRect (0, 0, VAL_TO_EDGE, VAL_TO_EDGE)`	Make the `Rect` the size of the object (for example, the canvas or bitmap).

Functions for Modifying Rects and Points

Use the following functions to set or modify the values in a `Rect` or `Point` structure.

- `RectSet` to set each of the four values of an existing Rect structure.
- `RectSetFromPoints` to set a Rect so that it defines the smallest rectangle that encloses two points.
- `RectSetBottom` to set the height of a Rect so that the bottom is a given value. (The bottom is *not* enclosed by the rectangle.)
- `RectSetRight` to set the height of a Rect so that the right edge is a given value. (The right edge is *not* enclosed by the rectangle.)
- `RectSetCenter` to set the top and left of a Rect so that it is centered around a give value, while keeping the same size.
- `RectOffset` to modify the top and left of a Rect so as to shift the location of the rectangle.
- `RectMove` to set the top and left of Rect to a given Point.
- `RectGrow` to modify the values in a Rect so that the rectangle grows or shrinks around its current center point.
- `PointSet` to set the two values in an existing Point structure.

Functions for Comparing or Obtaining Values from Rects and Points

Use the following functions to compare or obtain values from a `Rect` or `Point` structure.

- `RectBottom` to obtain the location of the bottom of a rectangle.
- `RectRight` to obtain the location of the right edge of a rectangle.
- `RectCenter` to obtain the location of the center of a rectangle.
- `RectEqual` to determine if two rectangles are identical.
- `RectEmpty` to determine if a rectangle is empty.
- `RectContainsPoint` to determine if a rectangle encloses a given point.
- `RectContainsRect` to determine if a rectangle completely encloses another rectangle.
- `RectSameSize` to determine if two rectangles are the same size.
- `RectUnion` to set a Rect to the smallest rectangle that encloses two given rectangles.

- `RectIntersection` to set a Rect to the largest rectangle that is enclosed by two given rectangles.
- `PointEqual` to determine if two points are at the same location.
- `PointPinnedToRect` to modify a Point structure, if needed, to ensure that it is within a give rectangle.

Using Bitmap Objects

A bitmap is a two-dimensional grid of pixels representing an image. There are some functions, such as `PlotBitmap` (for graph controls) and `DisplayImageFile` (for picture controls) which read an image out of a file and directly display it on a control.

There are other functions, however, which create or extract a bitmap, store them in memory, and return a bitmap ID. You can then use the bitmap ID in other functions.

Functions for Creating, Extracting, or Discarding Bitmap Objects

Use the following functions to create, extract, or discard bitmap objects.

- `NewBitmap` to create a bitmap object from scratch.
- `GetBitmapFromFile` to create a bitmap object using image data read from a file.
- `GetCtrlBitmap` to create a bitmap object from an image contained in a picture, picture ring, picture button, canvas, or graph control.
- `GetCtrlDisplayBitmap` to create a bitmap object from the current appearance of a control.
- `GetPanelDisplayBitmap` to create a bitmap object from the current appearance of a specified rectangular area of a panel.
- `ClipboardGetBitmap` to create a bitmap object from an image (if any) in the system clipboard
- `DiscardBitmap` to remove a bitmap from memory.

If you want to display images that are not rectangular or that have “holes” in them, you can use bitmaps that have a transparent background. If you are creating your bitmap image from scratch, you can achieve transparency by using the **mask** parameter to the `NewBitmap` function.

Windows Metafiles

A Windows metafile (`.WMF`) contains a description of an image that is scaleable without distortion. The description consists of a set of drawing commands rather than a bitmap. If you load a Window metafile image using `GetBitmapFromFile`, the image is converted to a bitmap. The size of the bitmap is the original size stored in the metafile.

If you try to display this bitmap on a canvas in an area of a different size, the image stretches or shrinks like any other bitmap. The scaling properties of the metafile are not used.

To use the scaling properties of the metafile, you must first load the metafile into a picture control. Use the following steps.

1. Create a picture control, and hide it.
2. Hide the frame of the picture control by setting `ATTR_FRAME_VISIBLE` to FALSE.
3. Set the picture control to the size in which you want to display the image on the canvas.
4. Set the `ATTR_FIT_MODE` attribute of the picture control to `VAL_SIZE_TO_PICTURE`.
5. Use `DisplayImageFile` to load the image into the picture control.
6. Use `GetCtrlBitmap` to obtain a bitmap object containing the image.

Windows metafiles do not always have background colors. When a metafile image is loaded using `GetBitmapFromFile`, the background is set to white. When a metafile image contained in a control is converted to a bitmap, the background color of the control is used.

Functions for Displaying or Copying Bitmap Objects

Use the following functions to display a bitmap object in a control or copy an image from a bitmap object to a control.

- `CanvasDrawBitmap` to display a bitmap in a canvas control.
- `SetCtrlBitmap` to set an image in a picture, picture ring, picture button, or graph control from a bitmap object. Can be used to replace an existing image create a new image.
- `ClipboardPutBitmap` to copy image data from a bitmap object to the system clipboard.

Functions for Retrieving Image Data from Bitmap Objects

Use the following functions to retrieve image data from bitmap objects.

- `GetBitmapInfo` to obtain size information about the image associated with a bitmap. This information can then be used in allocating the buffers to be passed to `GetBitmapData`.

- `AllocBitmapData` to allocate the buffers necessary for calling `GetBitmapData`. This is an alternative to calling `GetBitmapInfo` and allocating the buffers yourself.
- `GetBitmapData` to obtain the bit values that define the image associated with a bitmap.

Additions to Table 3-16, Graph and Strip Chart Attributes

The following new items belong in Table 3-16 of the *LabWindows/CVI User Interface Reference Manual*.

For graphs and strip charts		
`ATTR_INNER_LOG_MARKERS_VISIBLE`	`int`	Specifies whether labels and tick marks are shown next to the inner grid lines of log scale axes. (Default value: FALSE)
`ATTR_XAXIS_GAIN`	`double`	The factor used to scale the value labels on the X axis. For example, if the X value is 10.0 and `ATTR_XAXIS_GAIN` is 2.0, then the label on the X axis shows as 20.0. (Default value: 1.0)
`ATTR_XAXIS_OFFSET`	`double`	The amount added to the value labels on the X axis. For example, if the X value is 10.0 and `ATTR_XAXIS_OFFSET` is 5.0, then the label on the X axis shows as 15.0. The X value is multiplied by `ATTR_XAXIS_GAIN` before `ATTR_XAXIS_OFFSET` is added. (Default value: 0.0)
`ATTR_XUSE_LABEL_STRINGS`	`int`	Whether the X axis numerical value labels are replaced by strings associated with the X values. These strings can be specified either in the User Interface Editor or by calling the `InsertAxisItem` function.

continues

For graphs and strip charts (Continued)		
ATTR_YAXIS_GAIN	double	The factor used to scale the value labels on the Y axis. For example, if the Y value is 10.0 and ATTR_YAXIS_GAIN is 2.0, then the label on the Y axis shows as 20.0. (Default value: 1.0)
ATTR_YAXIS_OFFSET	double	The amount added to the value labels on the Y axis. For example, if the Y value is 10.0 and ATTR_YAXIS_OFFSET is 5.0, then the label on the Y axis shows as 15.0. The Y value is multiplied by ATTR_YAXIS_GAIN before ATTR_YAXIS_OFFSET is added. (Default value: 0.0)
ATTR_YAXIS_REVERSE	int	Whether to reverse the orientation of the Y axis so that the lowest value is shown at the top. If the orientation of the Y axis is reversed, the vertical orientation of the plots is also reversed.
ATTR_YUSE_LABEL_STRINGS	int	Whether the Y axis numerical value labels are replaced by strings associated with the Y values. These strings can be specified either in the User Interface Editor or by calling the InsertAxisItem function.

For graphs only		
ATTR_ACTIVE_YAXIS	int	Which of the two Y axes is used in plotting, setting a Y axis attribute, setting the Y axis range, or creating a graph cursor. Values: VAL_LEFT_YAXIS, VAL_RIGHT_YAXIS.
ATTR_ENABLE_ZOOMING	int	Whether the end-user can interactively zoom and pan the graph viewport. Default: FALSE
ATTR_XREVERSE	int	Whether to reverse the orientation of the X axis so that the lowest value is shown at the right. If the orientation of the X axis is reversed, the horizontal orientation of the plots is also reversed.

For graph cursors (GetCursorAttribute and SetCursorAttribute)		
ATTR_CURSOR_YAXIS	int	Used to change the Y axis to which a graph cursor is associated. When you create a graph cursor, its associated Y axis is determined by the value of ATTR_ACTIVE_YAXIS. Afterwards, the association can be changed using ATTR_CURSOR_YAXIS. The associated axis serves as the reference for the cursor position coordinates used in calls to SetGraphCursor and GetGraphCursor. Values: VAL_LEFT_YAXIS, VAL_RIGHT_YAXIS.

For graph plots (GetPlotAttribute and SetPlotAttribute)		
ATTR_PLOT_ORIGIN	int	Determines the placement of a text string or bitmap with respect to the coordinates specified in a call to PlotText or PlotBitmap. See discussion following this table.
ATTR_PLOT_SNAPPABLE	int	By default, graph cursors for which ATTR_CURSOR_MODE is VAL_SNAP_TO_POINT snap to the closest plot. To prevent cursors from snapping to a particular plot, set ATTR_PLOT_SNAPPABLE for the plot to FALSE.
ATTR_PLOT_YAXIS	int	Used to change the Y axis with which a plot is associated. When a plot is first plotted, the Y axis to which it is associated is determined by the value of ATTR_ACTIVE_YAXIS. Afterwards, the association can be changed using ATTR_PLOT_YAXIS. Values: VAL_LEFT_YAXIS, VAL_RIGHT_YAXIS.

Plot Origin Discussion

When PlotText or PlotBitmap is called, the text string or bitmap is placed on the graph with respect to a point specified by coordinates passed into the function. The orientation of the string or bitmap with respect to the point is determined by the ATTR_PLOT_ORIGIN attribute. The attribute specifies where the point (that is, the origin) is with respect to the rectangle that

implicitly encloses the string or bitmap. For example, VAL_LOWER_LEFT (the default value) specifies that the string or bitmap be plotted so that the lower left corner of its enclosing rectangle is at the point specified.

The possible values are shown in the following table.

Table 3-6. Values for ATTR_PLOT_ORIGIN

Value	Description
VAL_LOWER_LEFT	The lower left corner of the enclosing rectangle.
VAL_CENTER_LEFT	The midpoint of the left edge of the enclosing rectangle.
VAL_UPPER_LEFT	The upper left corner of the enclosing rectangle.
VAL_LOWER_CENTER	The midpoint of the bottom edge of the enclosing rectangle.
VAL_CENTER_CENTER	The center of the enclosing rectangle.
VAL_UPPER_CENTER	The midpoint of the top edge of the enclosing rectangle.
VAL_LOWER_RIGHT	The lower right corner of the enclosing rectangle.
VAL_CENTER_RIGHT	The midpoint of the right edge of the enclosing rectangle.
VAL_UPPER_RIGHT	The upper right corner of the enclosing rectangle.

Two Y Axis (graphs only)

There are always two Y axes on a graph. By default, only the left Y axis is visible.

You can make the Y axis visible by using the following code.

```
SetCtrlAttribute (panel, ctrl, ATTR_ACTIVE_YAXIS, VAL_RIGHT_AXIS);
SetCtrlAttribute (panel, ctrl, ATTR_YLABEL_VISIBLE, 1);
```

You can choose to make either one, both, or none of the Y axes visible.

The ATTR_ACTIVE_YAXIS attribute determines which of the two Y axes are used for the following actions.

- Adding a plot to the graph. (The active Y axis serves as the scaling reference.)
- Setting a Y axis attribute. (Each Y axis has its own attribute values.)
- Setting the Y axis range.
- Creating a graph cursor. (The cursor is associated with the active Y axis.)

Once a plot has been added to a graph, you can associate it with the other Y axis by using the ATTR_PLOT_YAXIS attribute.

Once a graph cursor has been created, you can associate it with the other Y axis by using the `ATTR_CURSOR_YAXIS` attribute. The associated Y axis serves as the reference for the cursor position coordinates in calls to `SetGraphCursor` and `GetGraphCursor`.

Changes to the Picture Control Image Bits functions

Image Bits Functions Superseded by New Functions

When picture controls were added to LabWindows/CVI you were able to manipulate bitmaps in picture controls using the following four functions.

```
GetImageBits
SetImageBits
GetImageInfo
AllocImageBits
```

Now, a more general set of functions for handling bitmap images as independent objects have been added. See the section *Functions for Creating, Extracting, or Discard Bitmap Objects*, earlier in this chapter. We recommend that you use these new functions with picture controls rather than the four functions listed here. Nevertheless, you can continue to use the original four functions.

24-Bit Pixel Depth Supported in Image Bits Functions

You can now use a 24-bit pixel depth in the four picture control image bits functions. When you do, the color table parameters to the functions are not used. The array of bits contains RGB values rather than indexes into the color table. Each RGB value in the array of bits represented by a 3-byte value of the form

```
0xRRGGBB
```

where RR, GG, and BB represent the red, green and blue intensity of the color. The RR byte should always be at the lowest memory address of the three bytes.

Using the System Attributes

The system attributes are set and obtained using the `SetSystemAttribute` and `GetSystemAttribute` functions. They are attributes that apply to the User Interface Library in general, rather than to particular instances of user interface objects. The following table lists the system attributes.

Table 3-7. System Attributes

Attribute	Type	Notes
ATTR_ALLOW_UNSAFE_TIMER_EVENTS	integer	1 - Allow unsafe timer events. 0 - Do not allow unsafe time events (the default) (Windows 95 and NT only). See discussion below.

Unsafe Timer Events

By default, timer control callbacks do not occur on MS Windows while you are moving or sizing a window, while the system menu is pulled down, or while the Alt-Tab key is pressed. (These conditions are called *event-blocking conditions*.) On Windows 95 and NT, you can use the ALLOW_UNSAFE_TIMER_EVENTS attribute to enable timer events under some, but not all, of the event-blocking conditions. If you set the ALLOW_UNSAFE_TIMER_EVENTS attribute to TRUE, timer events are blocked only under the following conditions.

- You have clicked on a window title bar, you are holding the mouse button down, but you are not moving the mouse.
- You are moving or resizing a window, and the Windows 95 "Show Window Contents While Dragging" option is disabled (or you are running on Windows NT.)

There are several limitations to this feature. One limitation is that while an event-blocking condition is in effect, timer callbacks are called no faster than once per 55 milliseconds.

Another limitation is that if a timer callback is called during an event-blocking condition and the callback causes events to be processed, mouse and keyboard input can behave erratically. Your program can cause this to happen in the following ways.

- The timer callback function calls ProcessSystemEvents in a loop.
- The timer callback function calls RunUserInterface, GetUserEvent, or a popup panel function such as MessagePopup or FileSelectPopup.
- Program execution is suspended in the timer callback function because of a breakpoint or run-time error.

In either case, the system functions normally once the timer callback returns.

This problem is inherent to Windows and occurs regardless of the development environment.

You should not enable this attribute until the code in your timer callbacks has been thoroughly debugged. The behavior of the system is undefined if you hit a breakpoint or run-time error when an event-blocking condition is in effect.

Additions to Table A-1, User Interface Library Error Codes

The following updates belong in Table A-1 of the *LabWindows/CVI User Interface Reference Manual*.

-124	Intensity plots cannot use transparent colors.
-125	Color is invalid.
-126	The specified callback function differs only by a leading underscore from another function or variable. Change one of the names for proper linking.
-127	Bitmap is invalid.
-128	There is no image in the control.

New User Interface Library Functions

AllocBitmapData

`int` **status** = **AllocBitmapData** (`int` **bitmapID**, `int **`**colorTable**, `char **`**bits**,
`unsigned char **`**mask**);

Purpose

Allocates the buffers necessary for calling `GetBitmapData` on a bitmap. If you use `GetBitmapInfo`, you must allocate the buffers yourself.

You must free the buffers when you are done with them.

Parameters

Input	**bitmapID**	integer	The ID of the bitmap object containing the image. The ID must have been obtained from `NewBitmap`, `GetBitmapFromFile`, `GetCtrlBitmap`, `ClipboardGetBitmap`, `GetCtrlDisplayBitmap`, or `GetPanelDisplayBitmap`.
Output	**colorTable**	pointer to integer	A pointer variable into which the address of the allocated color table buffer is placed.
	bits	pointer to unsigned char	A pointer variable into which the address of the allocated bits data buffer is placed.
	mask	pointer to unsigned char	A pointer variable into which the address of the allocated mask buffer is placed.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

You may pass NULL for any of the **colorTable**, **bits**, or **mask** parameters if you do not want the corresponding buffer to be allocated.

If the image does not exist, the **colorTable**, **bits**, and **mask** parameters are set to NULL. The **colorTable** parameter is set to NULL if the pixel depth of the image is greater than 8. The **mask** parameter is set to NULL if the image does not have a mask.

Warning: ***You must free the* colorTable, bitmap, *and* mask *buffers when you are done with them. Use the ANSI C Library*** `free` ***function.***

See Also

`GetBitmapData, GetBitmapInfo`

CanvasClear

`int` **status** = **CanvasClear** (`int` **panelHandle**, `int` **controlID**, `Rect` **rect**);

Purpose

Restores the specified rectangular area of a canvas control to the background color of the canvas control. The background color of the canvas control is determined by the `ATTR_PICT_BGCOLOR` attribute.

This operation is not restricted to the canvas clipping rectangle.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	rect	Rect	A `Rect` structure specifying the location and size of the rectangle to be cleared. Use `VAL_ENTIRE_OBJECT` to specify the entire canvas.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakeRect`

CanvasDefaultPen

`int` **status = CanvasDefaultPen** (`int` **panelHandle**, `int` **controlID**);

Purpose

Sets all of the attributes of the canvas pen to the default values. The defaults are shown in the following table.

Canvas Pen Attribute	Default Value
`ATTR_PEN_WIDTH`	1
`ATTR_PEN_STYLE`	`VAL_SOLID`
`ATTR_PEN_COLOR`	`VAL_BLACK`
`ATTR_PEN_FILL_COLOR`	`VAL_BLACK`
`ATTR_PEN_MODE`	`VAL_COPY_MODE`
`ATTR_PEN_PATTERN`	A solid pattern, expressed as an array of 8 unsigned characters, each of which is 0xFF.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`CanvasSetPenPosition, CanvasGetPenPosition, CanvasDrawLineTo`

CanvasDimRect

`int` **status** = **CanvasDimRect** (`int` **panelHandle**, `int` **controlID**, `Rect` **rect**);

Purpose

Overlays a checkerboard pattern in the specified rectangular area of a canvas control. This has the visual effect of dimming objects within the area.

The checkerboard pattern is drawn using current values of the following attribute.

`ATTR_PEN_FILL_COLOR`

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	rect	Rect	A `Rect` structure specifying the location and size of the area to be dimmed. Use `VAL_ENTIRE_OBJECT` to specify the entire canvas.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakeRect`

CanvasDrawArc

`int` **status = CanvasDrawArc** (`int` **panelHandle**, `int` **controlID**, `Rect` **rect**,
`int` **drawMode**, `int` **beginningAngle**, `int` **arcAngle**);

Purpose

Draws an arc on the canvas control. The arc is defined by specifying a rectangle that encloses the arc, along with a beginning angle (in tenths of degrees) and an arc angle (in tenths of degrees).

The arc is a section of an oval. A beginning angle of 0 indicates that the arc starts at the midpoint of the right edge of the rectangle. The arc angle indicates how far around the oval (counter-clockwise, up to 3600) the arc is drawn.

The frame of the arc is drawn using the current value of the following attributes:

`ATTR_PEN_COLOR`
`ATTR_PEN_MODE`
`ATTR_PEN_WIDTH`
`ATTR_PEN_STYLE` (ignored in Windows when pen width is greater than 1)

The interior of the arc is drawn using the current value of the following attributes.

`ATTR_PEN_FILL_COLOR`
`ATTR_PEN_MODE`
`ATTR_PEN_PATTERN`

The frame of the arc does not include the radius lines going from the center of the oval to the end points of the arc.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	rect	Rect	A `Rect` structure specifying the location and size of the rectangle within which to draw the arc.

continues

Parameters (Continued)

	drawMode	integer	Specifies whether the arc's frame or interior (or both) are drawn. Valid values: VAL_DRAW_FRAME VAL_DRAW_INTERIOR VAL_DRAW_FRAME_AND_INTERIOR
	beginningAngle	integer	The starting angle of the arc, in tenths of degrees. 0 indicates the arc starts at the midpoint of the right edge of the rectangle. 900 indicates that the arc starts at the midpoint of the top edge of the rectangle. Negative values are valid.
	arcAngle	integer	How far around the oval (counter-clockwise, up to 3600) the arc is drawn. Specified in tenths of degrees. Negative values are valid.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakeRect, CanvasDrawOval`

CanvasDrawBitmap

`int` **status** = **CanvasDrawBitmap** (`int` **panelHandle**, `int` **controlID**, `int` **bitmapID**, `Rect` **sourceRect**, `Rect` **destinationRect**);

Purpose

Draws a bitmap image (or portion thereof) in the specified destination rectangle on the canvas control.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	bitmapID	integer	The ID of the bitmap object containing the image. The ID must have been obtained from `NewBitmap`, `GetBitmapFromFile`, `GetCtrlBitmap`, `ClipboardGetBitmap`, `GetCtrlDisplayBitmap`, or `GetPanelDisplayBitmap`.
	sourceRect	Rect	A `Rect` structure specifying the portion of the bitmap to be drawn. The values are in terms of the pixel coordinates of the bitmap. The origin (0, 0) is at the upper left corner of the bitmap. Use `VAL_ENTIRE_OBJECT` to specify the entire image.
	destinationRect	Rect	A `Rect` structure specifying the size and location of the area in which the bitmap image is to be drawn on the canvas control. If **sourceRect** and **destinationRect** are not the same size, the bitmap is stretched or shrunk to fit.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

If you want the destination rectangle to be same size as the source rectangle, you can set the **height** and **width** in **destinationRect** to `VAL_KEEP_SAME_SIZE`.

If you want the bitmap to stretch to fit the size of the canvas, pass `VAL_ENTIRE_OBJECT` as **destinationRect**.

Example

The following code copies a bitmap image, without any stretching or shrinking, to the canvas control, starting 20 pixels below the top edge of the canvas, and 30 pixels to the right of left edge of the canvas.

```
CanvasDrawBitmap (panelHandle, controlID, bitmapID, VAL_ENTIRE_OBJECT,
MakeRect (20,30, VAL_KEEP_SAME_SIZE, VAL_KEEP_SAME_SIZE));
```

See also

`MakeRect`

CanvasDrawLine

`int` **status = CanvasDrawLine** (`int` **panelHandle**, `int` **controlID**, `Point` **start**, `Point` **end**);

Purpose

Draws a line between two specified points.

The line is drawn using the current value of the following attributes.

`ATTR_PEN_COLOR`
`ATTR_PEN_MODE`
`ATTR_PEN_WIDTH`
`ATTR_PEN_STYLE` (ignored in Windows when pen width is greater than 1)

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	start	Point	A Point structure specifying the location at which the line begins.
	end	Point	A Point structure specifying the location at which the line ends.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakePoint, CanvasDrawLineTo`

CanvasDrawLineTo

`int` **status = CanvasDrawLineTo** (`int` **panelHandle**, `int` **controlID**, `Point` **end**);

Purpose

Draws a line between the current pen position and a specified end point, and sets the pen position to the end point.

The line is drawn using the current value of the following attributes.

`ATTR_PEN_COLOR`
`ATTR_PEN_MODE`
`ATTR_PEN_WIDTH`
`ATTR_PEN_STYLE` (ignored in Windows when pen width is greater than 1)

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	end	Point	A Point structure specifying the location at which the line ends.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakePoint, CanvasGetPenPosition, CanvasSetPenPosition, CanvasDefaultPen, CanvasDrawLine`

CanvasDrawOval

```
int status = CanvasDrawOval (int panelHandle, int controlID, Rect rect,
                             int drawMode);
```

Purpose

Draws an oval on the canvas control within the specified rectangle.

The frame of the oval is drawn using the current value of the following attributes.

```
ATTR_PEN_COLOR
ATTR_PEN_MODE
ATTR_PEN_WIDTH
ATTR_PEN_STYLE    (ignored in Windows when pen width is greater than 1)
```

The interior of the oval is drawn using the current value of the following attributes:

```
ATTR_PEN_FILL_COLOR
ATTR_PEN_MODE
ATTR_PEN_PATTERN
```

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	rect	Rect	A `Rect` structure specifying the location and size of the rectangle within which to draw the oval.

	drawMode	integer	Specifies whether the oval's frame or interior (or both) are drawn. Valid values:
			`VAL_DRAW_FRAME`
			`VAL_DRAW_INTERIOR`
			`VAL_DRAW_FRAME_AND_INTERIOR`

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakeRect, CanvasDrawArc`

CanvasDrawPoint

`int` **status** = **CanvasDrawPoint** (`int` **panelHandle**, `int` **controlID**, `Point` **point**);

Purpose

Draws a point on the canvas control as the specified position.

The point is drawn using the current value of the following attributes:

```
ATTR_PEN_COLOR
ATTR_PEN_MODE
ATTR_PEN_WIDTH
```

At pen widths of greater than 1, the point may appear to be non-circular.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	point	Point	A Point structure specifying the location at which to draw the point.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakeRect, CanvasDrawArc`

CanvasDrawPoly

`int` **status = CanvasDrawPoly** (`int` **panelHandle**, `int` **controlID**, `int` **numberOfPoints**, `Point` **points**`[]`, `int` **wrap**, `int` **drawMode**);

Purpose

Draws a polygon on the canvas control by connecting the specified points.

The frame of the polygon is drawn using the current value of the following attributes.

`ATTR_PEN_COLOR`
`ATTR_PEN_MODE`
`ATTR_PEN_WIDTH`
`ATTR_PEN_STYLE` (ignored in Windows when pen width is greater than 1)

The interior of the polygon is drawn using the current value of the following attributes.

`ATTR_PEN_FILL_COLOR`
`ATTR_PEN_MODE`
`ATTR_PEN_PATTERN`

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	numberOfPoints	integer	The number of vertices in the polygon.
	points	Point array	An array of Point structures specifying the locations of the vertices of the polygon.
	wrap	integer	A nonzero value specifies that a line is drawn between last point and first point, thereby closing the polygon frame. This value is ignored when drawing only the interior.

	drawMode	integer	Specifies whether the polygon's frame or interior (or both) are drawn. Valid values:
			`VAL_DRAW_FRAME`
			`VAL_DRAW_INTERIOR`
			`VAL_DRAW_FRAME_AND_INTERIOR`

Return Value

status	integer	Refer to Appendix A for error codes.

CanvasDrawRect

`int` **status** = **CanvasDrawRect** (`int` **panelHandle**, `int` **controlID**, `Rect` **rect**, `int` **drawMode**);

Purpose

Draws a rectangle on the canvas control.

The frame of the rectangle is drawn using the current value of the following attributes.

```
ATTR_PEN_COLOR
ATTR_PEN_MODE
ATTR_PEN_WIDTH
ATTR_PEN_STYLE   (ignored in Windows when pen width is greater than 1)
```

The interior of the rectangle is drawn using the current value of the following attributes.

```
ATTR_PEN_FILL_COLOR
ATTR_PEN_MODE
ATTR_PEN_PATTERN
```

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	rect	Rect	A `Rect` structure specifying the location and size of the rectangle to be drawn.

	drawMode	integer	Specifies whether the rectangle's frame or interior (or both) are drawn. Valid values:
			`VAL_DRAW_FRAME`
			`VAL_DRAW_INTERIOR`
			`VAL_DRAW_FRAME_AND_INTERIOR`

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakeRect, CanvasDrawRoundedRect`

CanvasDrawRoundedRect

`int` **status** = **CanvasDrawRoundedRect** (`int` **panelHandle**, `int` **controlID**, `Rect` **rect**,
`int` **ovalHeight**, `int` **ovalWidth**,
`int` **drawMode**);

Purpose

Draws a rounded rectangle on the canvas control. Each corner of the rectangle is drawn as a quadrant of an oval.

The frame of the rectangle is drawn using the current value of the following attributes.

`ATTR_PEN_COLOR`
`ATTR_PEN_MODE`
`ATTR_PEN_WIDTH`
`ATTR_PEN_STYLE` (ignored in Windows when pen width is greater than 1)

The interior of the rectangle is drawn using the current value of the following attributes.

`ATTR_PEN_FILL_COLOR`
`ATTR_PEN_MODE`
`ATTR_PEN_PATTERN`

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	rect	Rect	A `Rect` structure specifying the location and size of the rectangle to be drawn.

continues

Parameters (Continued)

	ovalHeight	integer	The vertical diameter of the oval whose quadrants are drawn at the corners of the rounded rectangle.
	ovalWidth	integer	The horizontal diameter of the oval whose quadrants are drawn at the corners of the rounded rectangle.
	drawMode	integer	Specifies whether the rectangle's frame or interior (or both) are drawn. Valid values: VAL_DRAW_FRAME VAL_DRAW_INTERIOR VAL_DRAW_FRAME_AND_INTERIOR

Return Value

status	integer	Refer to Appendix A for error codes.

See also

MakeRect, CanvasDrawRect

CanvasDrawText

```
int status = CanvasDrawText (int panelHandle, int controlID, char text[],
                             char metaFont[], Rect bounds, int alignment);
```

Purpose

Draws a text string within a specified rectangular area on the canvas control. You can set the alignment of the string within the rectangle. If the string exceeds the size of the rectangle, it is clipped.

The text is drawn using the current value of the following attribute.

ATTR_PEN_COLOR

The background rectangle is drawn using the current value of the following attributes:

ATTR_PEN_FILL_COLOR
ATTR_PEN_MODE
ATTR_PEN_PATTERN

If you do not want the background rectangle to be drawn, set the ATTR_PEN_FILL_COLOR attribute of the canvas control to VAL_TRANSPARENT.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the LoadPanel, NewPanel, or DuplicatePanel function.
	controlID	integer	The defined constant (located in the .uir header file) which was assigned to the control in the User Interface Editor, or the ID returned by the NewCtrl or DuplicateCtrl function.
	text	string	The text string to be drawn within the rectangle.
	metaFont	string	Specifies the text font. Must be one of the predefined metafonts (see Table 3-5 in the *LabWindows/CVI User Interface Reference Manual*) or a metafont created by a call to CreateMetaFont.
	bounds	Rect	A Rect structure specifying location and size of the background rectangle within which the text is drawn.
	alignment	integer	Determines the placement of the text string within the background rectangle. See discussion below.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The values in the **bounds** parameter are in terms of pixel coordinates, with the origin (0,0) at the upper left corner of the canvas control.

If you want the size of the background rectangle to be adjusted automatically to the display size of the text string, set **height** and **width** in the **bounds** parameter to VAL_KEEP_SAME_SIZE.

The valid values for the **alignment** parameters are listed in the following table.

Value	Description
VAL_LOWER_LEFT	Draw the string in the lower left corner of the background rectangle.
VAL_CENTER_LEFT	Start the string from the midpoint of the left edge of the background rectangle.
VAL_UPPER_LEFT	Draw the string in the upper left corner of the background rectangle.
VAL_LOWER_CENTER	Center the string just above the bottom edge of the background rectangle.
VAL_CENTER_CENTER	Center the string in the middle of the background rectangle.
VAL_UPPER_CENTER	Center the string just below the top edge of the background rectangle.
VAL_LOWER_RIGHT	Draw the string in the lower right corner of the background rectangle.
VAL_CENTER_RIGHT	Draw the string so that it ends just at the midpoint of the right edge of the background rectangle.
VAL_UPPER_RIGHT	Draw the string in the upper right corner of the background rectangle.

If the background rectangle specified by **bounds** is smaller than the text display size, the text is clipped to the rectangle and the specified **alignment** is ignored. If the rectangle width is smaller than the text display width, the text is displayed from the left. If the rectangle height is smaller than the text display height, the text is displayed from the top.

See also

MakeRect, CanvasDrawTextAtPoint

CanvasDrawTextAtPoint

`int` **status = CanvasDrawTextAtPoint** (`int` **panelHandle**, `int` **controlID**, `char` **text**`[]`, `char` **metaFont**`[]`, `Point` **anchorPoint**, `int` **alignment**);

Purpose

Draws a text string at the specified location in the canvas control. The location is in terms of an anchor point and an alignment around the point.

The text is drawn using the current value of the following attribute.

`ATTR_PEN_COLOR`

The background of the text is drawn using the current value of the following attributes:

```
ATTR_PEN_FILL_COLOR
ATTR_PEN_MODE
ATTR_PEN_PATTERN
```

If you do not want the background rectangle to be drawn, set the `ATTR_PEN_FILL_COLOR` attribute of the canvas control to `VAL_TRANSPARENT`.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	text	string	The text string to be drawn at the anchor point.
	metaFont	string	Specifies the text font. Must be one of the predefined metafonts (see Table 3-5 in the *LabWindows/CVI User Interface Reference Manual*) or a metafont created by a call to `CreateMetaFont`.
	anchorPoint	Point	A Point structure specifying location of the point at which the text is drawn.
	alignment	integer	Determines the placement of the text string in relation to the anchor point. See discussion below.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Each **alignment** value refers to a point on the rectangle that implicitly encloses the text string. The text string is placed so that the point indicated by the **alignment** parameter is at the location specified the **anchorPoint** parameter. The valid values for the **alignment** parameter are listed in the following table.

Value	Description
`VAL_LOWER_LEFT`	Draw the string so that lower left corner of its enclosing rectangle is at the location specified by **anchorPoint**.
`VAL_CENTER_LEFT`	Draw the string so that midpoint of the left edge of its enclosing rectangle is at the location specified by **anchorPoint**.
`VAL_UPPER_LEFT`	Draw the string so that upper left corner of its enclosing rectangle is at the location specified by **anchorPoint**.
`VAL_LOWER_CENTER`	Draw the string so that midpoint of the bottom edge of its enclosing rectangle is at the location specified by **anchorPoint**.
`VAL_CENTER_CENTER`	Draw the string so that center of its enclosing rectangle is at the location specified by **anchorPoint**.
`VAL_UPPER_CENTER`	Draw the string so that midpoint of the top edge of its enclosing rectangle is at the location specified by **anchorPoint**.
`VAL_LOWER_RIGHT`	Draw the string so that lower right corner of its enclosing rectangle is at the location specified by **anchorPoint**.
`VAL_CENTER_RIGHT`	Draw the string so that midpoint of the right edge of its enclosing rectangle is at the location specified by **anchorPoint**.
`VAL_UPPER_RIGHT`	Draw the string so that upper right corner of its enclosing rectangle is at the location specified by **anchorPoint**.

See also

`MakeRect, CanvasDrawText`

CanvasEndBatchDraw

`int` **nestingDepth = CanvasEndBatchDraw** (`int` **panelHandle**, `int` **controlID**);

Purpose

Ends the batch drawing started with `CanvasStartBatchDraw`.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

Return Value

nestingDepth	integer	The number of calls to `CanvasStartBatchDraw` that have not been matched by calls to `CanvasEndBatchDraw` (including this call). A negative value indicates that an error occurred. Refer to Appendix A for error codes.

See also

`CanvasStartBatchDraw`

CanvasGetClipRect

`int` **status** = **CanvasGetClipRect** (`int` **panelHandle**, `int` **controlID**, `Rect *`**clipRect**);

Purpose

Obtains the current clipping rectangle for the canvas control. All drawing operations are restricted to the area in the clipping rectangle. Any drawing outside the clipping rectangle is not shown. Exception: `CanvasClear` is not restricted to the clipping rectangle.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
Output	**clipRect**	Rect	The `Rect` structure into which the location and size of the clipping rectangle are stored. If clipping is disabled (the default state), the height and width values in the structure are set to zero.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`CanvasSetClipRect`

CanvasGetPenPosition

`int` **status** = **CanvasGetPenPosition** (`int` **panelHandle**, `int` **controlID**, `Point *`**point**);

Purpose

Obtains the current position of the canvas pen.

Note: `CanvasDrawLineTo` ***is the only canvas drawing function that uses the pen position.***

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
Output	**point**	Point	The `Point` structure into which the current position of the canvas pen is stored.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`CanvasSetPenPosition, CanvasDefaultPen, CanvasDrawLineTo`

CanvasGetPixel

`int` **status = CanvasGetPixel** (`int` **panelHandle**, `int` **controlID**, `Point` **pixelPoint**, `int *`**pixelColor**);

Purpose

Obtains the color of a single pixel on a canvas control.

Note: ***The canvas control maintains an internal bitmap reflecting all of the drawing operations (except for drawing operations made while the*** `ATTR_DRAW_POLICY` ***attribute is set to*** `VAL_DIRECT_TO_SCREEN`***). There are times during which the internal bitmap contains the result of recent drawing operations that have not yet been reflected on the screen. This function obtains the pixel colors from the internal bitmap, not from the screen.***

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	pixelPoint	Point	A Point structure indicating the location of a pixel. The location is in terms of unscaled pixel coordinates. The origin (0,0) is the upper left corner of the canvas control.
Output	**pixelColor**	integer	The RGB color value of the pixel at the specified point.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakePoint, CanvasGetPixels`

CanvasGetPixels

```
int status = CanvasGetPixels (int panelHandle, int controlID, Rect rect,
                              int pixelColors[]);
```

Purpose

Obtains the colors of the pixels in the specified rectangular area of a canvas control.

Note: ***The canvas control maintains an internal bitmap reflecting all of the drawing operations (except for drawing operations made while the*** `ATTR_DRAW_POLICY` ***attribute is set to*** `VAL_DIRECT_TO_SCREEN`***). There are times during which the internal bitmap contains the result of recent drawing operations that have not yet been reflected on the screen. This function obtains the pixel colors from the internal bitmap, not from the screen.***

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	rect	Rect	The `Rect` structure specifying the location and size of the rectangular area from which to obtain the pixel colors. The location and size are expressed in terms of unscaled pixel coordinates. The origin (0,0) is the upper left corner of the canvas control. Use `VAL_ENTIRE_OBJECT` to specify the entire canvas.
Output	**pixelColors**	integer array	An array of RGB color values of the pixels in the specified rectangle. See discussion below.

Parameter Discussion

The total number of elements in the **pixelColors** array should be equal to **rect.height * rect.width**. The pixel color values are stored in row-major order. For example, if **rect** has the following values,

rect.top	50
rect.left	60
rect.height	20
rect.width	15

then the color of pixel {x = 65, y = 58} is stored in pixel array at the following index.

(y - **rect.top**) * **rect.width** + (x - **rect.left**)

= (58-50)*15 + (65-60)

= 125

When using a **rect.width** of `VAL_TO_EDGE`, substitute the following for **rect.width** in the above formula.

(total width of canvas) - **rect.left**

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakePoint, CanvasGetPixel`

CanvasInvertRect

`int` **status** = **CanvasInvertRect** (`int` **panelHandle**, `int` **controlID**, `Rect` **rect**);

Purpose

Inverts the colors in the specified rectangular area of a canvas control. The colors that result from the inversion are dependent on the operating system. If you invert the same rectangle twice, you are guaranteed to get the original colors back.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	rect	Rect	A `Rect` structure which specifies the location and size of the rectangular area in which to invert the colors. Use `VAL_ENTIRE_OBJECT` to specify the entire canvas.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakeRect`

CanvasScroll

`int` **status** = **CanvasScroll** (`int` **panelHandle**, `int` **controlID**, `Rect` **scrollRect**,
`int` **scrollAmountInXDirection**, `int` **scrollAmountInYDirection**);

Purpose

Scrolls the contents of the specified rectangular area of a canvas control. The area that is exposed by the scrolling is filled using the current value of the `ATTR_PEN_FILL_COLOR` attribute. The contents of the canvas outside the specified rectangular area is not affected by the scrolling.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	scrollRect	Rect	A `Rect` structure which specifies the location and size of the rectangular area to scroll. Use `VAL_ENTIRE_OBJECT` to specify the entire canvas.
	scrollAmountInXDirection	integer	The amount to scroll horizontally.
	scrollAmountInYDirection	integer	The amount to scroll vertically.

Parameter Discussion

A positive value for **scrollAmountInXDirection** moves the contents of the rectangle to the right. An area on the left side of the rectangle is thereby exposed. It is filled with the current value of the `ATTR_PEN_FILL_COLOR` attribute.

A negative value for **scrollAmountInXDirection** moves the contents of the rectangle to the left. An area on the right side of the rectangle is thereby exposed. It is filled with the current value of the `ATTR_PEN_FILL_COLOR` attribute.

A positive value for **scrollAmountInYDirection** moves the contents of the rectangle down. An area at the top of the rectangle is thereby exposed. It is filled with the current value of the ATTR_PEN_FILL_COLOR attribute.

A negative value for **scrollAmountInYDirection** moves the contents of the rectangle the up. An area at the bottom of the rectangle is thereby exposed. It is filled with the current value of the ATTR_PEN_FILL_COLOR attribute.

See Also

MakeRect

CanvasSetClipRect

int **status** = **CanvasSetClipRect** (int **panelHandle**, int **controlID**, Rect **clipRect**);

Purpose

Sets the clipping rectangle for the canvas control. All drawing operations are restricted to the area in the clipping rectangle. Any drawing outside the clipping rectangle is not shown. Exception: CanvasClear is not restricted to the clipping rectangle.

Changing the clipping rectangle does not affect current contents of the canvas.

In the initial state for a canvas control, clipping is disabled.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the LoadPanel, NewPanel, or DuplicatePanel function.
	controlID	integer	The defined constant (located in the .uir header file) which was assigned to the control in the User Interface Editor, or the ID returned by the NewCtrl or DuplicateCtrl function.
	clipRect	Rect	A Rect structure specifying into the location and size of the clipping rectangle. To disable clipping, set the height and width of **clipRect** to zero, or use VAL_EMPTY_RECT.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`CanvasGetClipRect`

CanvasSetPenPosition

`int` **status** = **CanvasSetPenPosition** (`int` **panelHandle**, `int` **controlID**, `Point` **point**);

Purpose

Sets the position of the canvas pen.

Note: `CanvasDrawLineTo` ***is the only canvas drawing function that uses the pen position.***

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	point	Point	A Point structure specifying the new position of the canvas pen.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`CanvasGetPenPosition, CanvasDefaultPen, CanvasDrawLineTo`

CanvasStartBatchDraw

`int` **nestingDepth** = **CanvasStartBatchDraw** (`int` **panelHandle**, `int` **controlID**);

Purpose

This function can be used to increase the drawing performance of the canvas control. In general, it should be used whenever you want to make two or more consecutive calls to canvas drawing functions. Each call to `CanvasStartBatchDraw` should be matched with a call to `CanvasEndBatchDraw`.

Before drawing operations can be performed, certain operating system functions must be invoked to prepare for drawing on the particular canvas. Without batch drawing, these system functions must be called for each canvas drawing operation. With batch drawing, the system functions are called only once for all of the drawing operations between `CanvasStartBatchDraw` and the matching `CanvasEndBatchDraw`.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

Return Value

nestingDepth	integer	The number of calls to `CanvasStartBatchDraw` (including this call) that have not been matched by calls to `CanvasEndBatchDraw`. A negative value indicates that an error occurred. Refer to Appendix A for error codes.

Using This Function

The following code shows an example of how you would incorporate a sequence of drawing operations on the same canvas control into one batch.

```
CanvasStartBatchDraw (panelHandle, controlID);
CanvasDrawLine (panelHandle, controlID, point1, point2);
CanvasDrawLine (panelHandle, controlID, point3, point4);
CanvasDrawRect (panelHandle, controlID, rect5);
CanvasEndBatchDraw (panelHandle, controlID);
```

During a batch draw, you may call drawing operations on other canvas controls or call other User Interface Library functions that perform drawing operations or process events. This has the effect of implicitly ending the batch. The next time you call a drawing function on the same canvas, the batch is implicitly restarted.

You may nest calls to `CanvasStartBatchDraw`.

Failure to properly match `CanvasStartBatchDraw` and `CanvasEndBatchDraw` calls can negate the potential performance improvements but does not cause any other ill effects.

Note: ***If the*** `ATTR_DRAW_POLICY` ***attribute for the canvas control is set to*** `VAL_UPDATE_IMMEDIATELY`***, no update to the screen occurs until the batch is ended. Also, changing values of the*** `ATTR_DRAW_POLICY` ***and*** `ATTR_OVERLAP_POLICY` ***attributes during a batch draw has no effect until after the batch is ended.***

See also

`CanvasEndBatchDraw`

CanvasUpdate

`int` **status = CanvasUpdate** (`int` **panelHandle**, `int` **controlID**, `Rect` **updateArea**);

Purpose

Immediately updates on the screen the contents of the canvas control within the specified rectangular area.

The canvas control maintains an internal bitmap reflecting all of the drawing operations (except for drawing operations made while the `ATTR_DRAW_POLICY` attribute is set to `VAL_DIRECT_TO_SCREEN`). Maintaining the internal bitmap ensures that the canvas is redrawn correctly when it is exposed.

This function copies the content of the specified area in the internal bitmap to the canvas control.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	updateArea	Rect	A `Rect` structure specifying the location and size of the rectangular to be updated from the internal bitmap. Use `VAL_ENTIRE_OBJECT` to specify the entire canvas control.

Return Value

status	integer	Refer to Appendix A for error codes.

See also

`MakeRect`

ClearAxisItems

`int` **status** = **ClearAxisItems** (`int` **panelHandle**, `int` **controlID**, `int` **axis**);

Purpose

This function deletes all string/value pairs from the list of label strings for a graph or strip chart axis.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

	axis	integer	Specifies the axis from which to delete all of the string/value pair(s). Valid values:
			`VAL_XAXIS`
			`VAL_LEFT_YAXIS`
			`VAL_RIGHT_YAXIS` (graphs only)

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`InsertAxisItem, ReplaceAxisItem, DeleteAxisItem, GetNumAxisItems`

ClipboardGetBitmap

`int` **status** = **ClipboardGetBitmap** (`int *`**bitmapID**, `int *`**available**);

Purpose

Determines whether or not a bitmap image is available on the system clipboard and optionally retrieves a copy of the bitmap. The bitmap ID can then be passed to any function that accepts a bitmap, such as `CanvasDrawBitmap`.

You can discard the bitmap by passing its ID to `DiscardBitmap`.

Parameters

Output	**bitmapID**	integer	An ID that serves as a handle to the bitmap copied from the clipboard. It is set to NULL if there is no bitmap on the clipboard. If you not want a copy of the bitmap, pass NULL.
	available	integer	Is set to 1 if a bitmap is available on the system clipboard, 0 otherwise. You may pass NULL for this parameter.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`ClipboardPutBitmap, ClipboardGetText, GetBitmapData, SetCtrlBitmap, PlotBitmap, CanvasDrawBitmap, DiscardBitmap.`

ClipboardGetText

`int` **status = ClipboardGetText** (`char **`**text**, `int *`**available**);

Purpose

Determines whether or not a text string is available on the system clipboard and optionally retrieves a copy of the text.

When the copy of the text is no longer needed, you should free it by calling the `free` function.

Parameters

Output	**text**	string	A pointer to the nul-terminated string copied from the clipboard. It is set to NULL if there is no text on the clipboard. If you not want a copy of the text, pass NULL.
	available	integer	Is set to 1 if a text string is available on the system clipboard, 0 otherwise. You may pass NULL for this parameter.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`ClipboardPutText, ClipboardGetBitmap.`

ClipboardPutBitmap

`int` **status** = **ClipboardPutBitmap** (`int` **bitmapID**);

Purpose

Copies a bitmap image onto the system clipboard.

Parameter

Input	**bitmapID**	integer	The ID of the bitmap object containing the image. The ID must have been obtained from `NewBitmap`, `GetBitmapFromFile`, `GetCtrlBitmap`, `ClipboardGetBitmap`, `GetCtrlDisplayBitmap`, or `GetPanelDisplayBitmap`.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`ClipboardGetBitmap, ClipboardPutText, NewBitmap, GetBitmapFromFile, GetCtrlBitmap, GetCtrlDisplayBitmap, GetPanelDisplayBitmap, ClipboardGetBitmap.`

ClipboardPutText

`int` **status** = **ClipboardPutText** (`char` **text**`[]`);

Purpose

Copies a text string onto the system clipboard.

Parameter

Input	**text**	string	A nul-terminated string.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`ClipboardGetText, ClipboardPutBitmap`

DeleteAxisItem

`int` **status** = **DeleteAxisItem** (`int` **panelHandle**, `int` **controlID**, `int` **axis**,
`int` **itemIndex**, `int` **numberOfItems**);

Purpose

Deletes one or more string/value pairs from the list of label strings for a graph or strip chart axis. These strings appear in place of the numerical labels. They appear at the location of their associated values on the graph or strip chart.

To see string labels on an X axis, you must set the `ATTR_XUSE_LABEL_STRINGS` attribute to TRUE. To see string labels on a Y axis, you must set the `ATTR_YUSE_LABEL_STRINGS` attribute to TRUE.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

	axis	integer	Specifies the axis from which to delete the selected string/value pair(s). Valid values:
			`VAL_XAXIS`
			`VAL_LEFT_YAXIS`
			`VAL_RIGHT_YAXIS` (graphs only)

	itemIndex	integer	The zero-based index of the first item to be deleted.
	numberOfItems	string	The number of items to delete.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`InsertAxisItem, ReplaceAxisItem, ClearAxisItems, GetNumAxisItems`

DiscardBitmap

`int` **status = DiscardBitmap** (`int` **bitmapID**);

Purpose

Discards a bitmap object.

Parameter

Input	**bitmapID**	integer	The ID of the bitmap object. The ID must have been obtained from `NewBitmap`, `GetBitmapFromFile`, `GetCtrlBitmap`, `ClipboardGetBitmap`, `GetCtrlDisplayBitmap`, or `GetPanelDisplayBitmap`.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`NewBitmap, GetBitmapFromFile, GetCtrlBitmap, GetCtrlDisplayBitmap, GetPanelDisplayBitmap, ClipboardGetBitmap.`

Get3dBorderColors

```
int status = Get3dBorderColors (int baseColor, int *highlightColor,
                                int *lightColor, int *shadowColor,
                                int *darkShadowColor);
```

Purpose

Takes an RGB value for the base color of an object and returns the RGB values for colors that can be used to make the object look 3-dimensional. The colors returned are similar to the colors used in Windows 95 for drawing 3-dimensional objects.

Parameters

Input	**baseColor**	integer	The RGB value for the color of an object.
Output	**highlightColor**	integer	The RGB value for the color used to indicate the edges of the object that are in the most direct light.
	lightColor	integer	The RGB value for the color used to indicate the transition between the highlight color and the base color of the object.
	shadowColor	integer	The RGB value for the color used to indicate the edges of the object that are angled away from the light.
	darkShadowColor	integer	The RGB value for the color used to indicate the edges of the object that are angled farthest away from the light.

Parameter Discussion

You may pass NULL for any of the output parameters.

Currently, the **lightColor** is always the same as the **baseColor**, as is the case in Windows 95.

Currently, the **darkShadowColor** this is always black, as is the case in Windows 95.

GetAxisItem

```
int status = GetAxisItem (int panelHandle, int controlID, int axis,
                          int itemIndex, char itemLabel[], double *itemValue);
```

Purpose

This function retrieves the string/value pair at the specified index in the list of label strings for a graph or strip chart axis.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

	axis	integer	Specifies the axis from which to retrieve the selected string/value pair. Valid values:
			`VAL_XAXIS`
			`VAL_LEFT_YAXIS`
			`VAL_RIGHT_YAXIS` (graphs only)

	itemIndex	integer	A zero-based index into the list of label strings.
Output	**itemLabel**	string	Buffer in which the label string is returned.
	itemValue	double	The value associated with the label string.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

itemLabel must be large enough to hold the label string, including the terminating NUL byte. You can use `GetAxisItemLabelLength` to determine the length of the label string.

You may pass NULL for either of the output parameters.

See Also

`InsertAxisItem, GetNumAxisItems, GetAxisItemLabelLength`

GetAxisItemLabelLength

`int` **status** = **GetAxisItemLabelLength** (`int` **panelHandle**, `int` **controlID**, `int` **axis**, `int` **itemIndex**, `int` ***length**);

Purpose

This function obtains the number of characters in a label string for a graph or strip chart axis. The label string is specified by its index in the list of string/value pairs for that axis.

The length returned does not include the terminating NUL byte.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

	axis	integer	Specifies the axis for which to return the length of the selected label string. Valid values:
			`VAL_XAXIS`
			`VAL_LEFT_YAXIS`
			`VAL_RIGHT_YAXIS` (graphs only)

	itemIndex	integer	A zero-based index into the list of label strings.
Output	**length**	string	The length of the selected label string. Excludes the terminating NUL byte.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

itemLabel must be large enough to hold the label string, including the terminating NUL byte. You can use `GetAxisItemLabelLength` to determine the length of the label string.

You may pass NULL for either of the output parameters.

See Also

`InsertAxisItem, GetNumAxisItems, GetAxisItem.`

GetAxisScalingMode

`int` **status = GetAxisScalingMode** (`int` **panelHandle**, `int` **controlID**, `int` **axis**, `int *`**axisScaling**, `double *`**min**, `double *`**max**);

Purpose

Obtains the scaling mode and the range of any graph axis or the Y axis of a strip chart.

This function is not valid for the X axis of a strip chart. To obtain the X offset and X increment for a strip chart, use the `GetCtrlAttribute` function with the `ATTR_XAXIS_OFFSET` and `ATTR_XAXIS_GAIN` attributes.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	axis	integer	Specifies for which axis to obtain the mode and range. Valid values: `VAL_XAXIS` (graphs only) `VAL_LEFT_YAXIS` (graphs and strip charts) `VAL_RIGHT_YAXIS` (graphs only)
Output	**axisScaling**	integer	The scaling mode used for the axis. See table below.
	min	double	The current minimum value on the axis.
	max	double	The current maximum value on the axis.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

axisScaling is one of the following values.

Valid Value	Description
VAL_MANUAL	The axis is set to manual scaling, and its range is defined by **min** and **max**.
VAL_AUTOSCALE	The axis is set to auto scaling. **min** and **max** are not used. Auto scaling is not valid for strip charts.
VAL_LOCK	The axis is set to manual scaling using the current (possibly auto scaled) minimum and maximum values on the axis. VAL_LOCK is not valid for strip charts.

If you call SetAxisScalingMode with **axisScaling** set to VAL_AUTOSCALE, and you then call SetAxisScalingMode with **axisScaling** set to VAL_LOCK, GetAxisScalingMode returns **axisScaling** as VAL_MANUAL.

max always exceeds **min**.

You may pass NULL for any of the output parameters.

See Also

SetAxisScalingMode

GetBitmapData

int **status** = **GetBitmapData** (int **bitmapID**, int ***bytesPerRow**, int ***pixelDepth**,
int ***width**, int ***height**, int **colorTable**[],
unsigned char **bits**[], unsigned char **mask**[]);

Purpose

Obtains the bit values that define the image associated with a bitmap. Before calling GetBitmapData, you must do one of the following.

Call GetBitmapInfo to get the size of the buffers needed, and then allocate the buffers, or

Call AllocBitmapData.

Parameters

Input	**bitmapID**	integer	The ID of the bitmap object containing the image. The ID must have been obtained from `NewBitmap`, `GetBitmapFromFile`, `GetCtrlBitmap`, `ClipboardGetBitmap`, `GetCtrlDisplayBitmap`, or `GetPanelDisplayBitmap`.
Output	**bytesPerRow**	integer	The number of bytes on each scan line of the image.
	pixelDepth	integer	The number of bits per pixel.
	width	integer	The width of the image, in pixels.
	height	integer	The height of the image, in pixels.
	colorTable	integer array	An array of RGB color values, or NULL if **pixelDepth** is greater than 8.
	bits	unsigned char array	An array of bits that determine the colors to be displayed on each pixel in the image.
	mask	unsigned char array	An array containing one bit per pixel in the image. Each bit specifies whether the pixel is actually drawn. May be NULL.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

If there is no image, the **width** and **height** parameters are set to -1. If the bitmap originated from a Windows metafile (`.WMF`), the size of the bitmap obtained by this function is the size stored in the original Windows metafile.

The **pixelDepth** parameter is set to 1, 2, 8, or 24.

The number of bits in the **bits** array for each pixel is equal to the **pixelDepth** value. If **pixelDepth** is 8 or less, the **bits** array is filled with indices into the **colorTable** array, and the number of entries in the **colorTable** array is 2 raised to the power of the **pixelDepth** parameter. If **pixelDepth** is greater than 8, the **colorTable** array is not used, and the **bits** array contains the actual RGB values. For a **pixelDepth** of 24, each pixel is represented by a 3-byte RGB value of the form `0xRRGGBB`, where `RR`, `GG`, and `BB` represent the red, green and blue intensity of the color. The `RR` byte is always at the lowest memory address of the three bytes.

The first pixel in the **bits** array is at the top, left corner of the image. The pixels in the array are in row-major order.

If `GetBitmapInfo` sets the **colorSize** parameter to zero, the **colorTable** array is not filled in.

In the **mask** array, a bit value of 1 indicates that the pixel is drawn. 0 indicates that the pixel is not drawn. Exception: If an image has a **pixelDepth** of 1, pixels with a **bits** value of 1 (foreground pixels) are always drawn and the **mask** affects only the pixels with a bitmap of 0 (background pixels). Each row of the mask is padded to the nearest even-byte boundary. For example, if the width of the image is 21 pixels, then there are 32 bits (in other words, 4 bytes) of data in each row of the mask.

A mask is useful for achieving transparency.

If `GetBitmapInfo` sets the **maskSize** parameter to zero, the **mask** array is not filled in.

Note: ***Only images that are 32 pixels wide and 32 pixels high can use a mask.***

You may pass NULL for any of the output parameters.

See Also

`NewBitmap, GetBitmapFromFile, GetCtrlBitmap, GetCtrlDisplayBitmap, GetPanelDisplayBitmap, ClipboardGetBitmap, GetBitmapInfo, AllocBitmapData.`

GetBitmapFromFile

`int` **status** = **GetBitmapFromFile** (`char` **fileName**`[]`, `int *`**bitmapID**);

Purpose

Reads a bitmap image from a file and creates a bitmap object. The bitmap ID can then be passed to any function that accepts a bitmap, such as `CanvasDrawBitmap` or `ClipboardPutBitmap`.

You can discard the bitmap object by passing its ID to `DiscardBitmap`.

You can use the following image types:

`PCX:` Windows and SPARCstation
`BMP, DIB, RLE, ICO, WMF:` Windows only
`XWD:` SPARCstation only

Parameters

Input	**fileName**	string	The pathname of the file which contains the image. If the name is a simple filename, the file is loaded from the project. If it is not found in the project, the file is loaded from the directory containing the project.
Output	**bitmapID**	integer	An ID that serves as a handle to the bitmap object.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`ClipboardPutBitmap, GetBitmapData, SetCtrlBitmap, PlotBitmap, CanvasDrawBitmap, DiscardBitmap.`

GetBitmapInfo

`int` **status = GetBitmapInfo** (`int` **bitmapID**, `int *`**colorSize**, `int *`**bitsSize**, `int *`**maskSize**);

Purpose

Obtains size information about the image associated with a bitmap. This information can then be used in allocating the buffers to be passed to the `GetBitmapData` function.

As an alternative to this function, you can call `AllocBitmapData`, which allocates the buffers for you.

Parameters

Input	**bitmapID**	integer	The ID of the bitmap object containing the image. The ID must have been obtained from `NewBitmap`, `GetBitmapFromFile`, `GetCtrlBitmap`, `ClipboardGetBitmap`, `GetCtrlDisplayBitmap`, or `GetPanelDisplayBitmap`.
Output	**colorSize**	integer	The number of bytes in the image color table (0 if the pixel depth of the image is greater than 8).
	bitsSize	integer	The number of bytes in the image bitmap.
	maskSize	integer	The number of bytes in the image mask (0 if there is no mask).

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

You may pass NULL for any of the output parameters.

See Also

`GetBitmapData, AllocBitmapData.`

GetCtrlBitmap

`int` **status** = **GetCtrlBitmap** (`int` **panelHandle**, `int` **controlID**, `int` **imageID**,
`int` *__bitmapID__);

Purpose

Obtains a bitmap image from a control and stores it in a bitmap object. The bitmap ID can then be passed to any function that accepts a bitmap, such as `CanvasDrawBitmap` or `ClipboardPutBitmap`.

The following control types can contain images:

picture controls
picture rings
picture buttons
graph controls
canvas controls

You can use this function on images set using `DisplayImageFile`, `InsertListItem`, `ReplaceListItem`, `PlotBitmap`, or `SetImageBits`, or `SetCtrlAttribute` with the `ATTR_IMAGE_FILE` attribute.

You can discard the bitmap object by passing its ID to `DiscardBitmap`.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	imageID	integer	For picture rings, the zero-based index of an image in the ring. For graphs, this argument is the **plotHandle** returned from `PlotBitmap`. For picture controls, picture buttons, and canvas controls, this argument is ignored.
Output	**bitmapID**	integer	An ID that serves as a handle to the bitmap object.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`ClipboardPutBitmap, GetBitmapData, SetCtrlBitmap, PlotBitmap, CanvasDrawBitmap, DiscardBitmap.`

GetCtrlDisplayBitmap

`int` **status = GetCtrlDisplayBitmap** (`int` **panelHandle**, `int` **controlID**, `int` **includeLabel**, `int *` **bitmapID**);

Purpose

This function creates a bitmap object containing a "snapshot" image of the current appearance of the specified control. The bitmap ID can then be passed to any function that accepts a bitmap, such as `CanvasDrawBitmap` or `ClipboardPutBitmap`.

For example, you can paste a picture of a control onto the system clipboard by calling `GetCtrlDisplayBitmap` and then passing the bitmap ID to `ClipboardPutBitmap`.

You can discard the bitmap object by passing the ID to the `DiscardBitmap` function.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	includeLabel	integer	If nonzero, the control label is included in the image.
Output	**bitmapID**	integer	An ID that serves as a handle to the bitmap object.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`ClipboardPutBitmap, GetBitmapData, GetPanelDisplayBitmap, SetCtrlBitmap, PlotBitmap, CanvasDrawBitmap, DiscardBitmap.`

GetNumAxisItems

`int` **status = GetNumAxisItems** (`int` **panelHandle**, `int` **controlID**, `int` **axis**, `int *`**count**);

Purpose

Returns the number of items in the list of label strings for a graph or strip chart axis.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

	axis	integer	Specifies the axis for which to return the count of string/value pairs. Valid values:
			`VAL_XAXIS`
			`VAL_LEFT_YAXIS`
			`VAL_RIGHT_YAXIS` (graphs only)

Output	**count**	integer	The number of string/value pairs for the axis.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`InsertAxisItem, GetAxisItem, GetAxisItemLabelLength`

GetPanelDisplayBitmap

```
int status = GetPanelDisplayBitmap (int panelHandle, int scope, Rect area,
                                    int *bitmapID);
```

Purpose

This function creates a bitmap object containing a "snapshot" image of the current appearance of the specified panel. The bitmap ID can then be passed any function that accepts a bitmap, such as `CanvasDrawBitmap` or `ClipboardPutBitmap`.

For example, you can paste a picture of a panel onto the system clipboard by calling `GetPanelDisplayBitmap` and then passing the bitmap ID to `ClipboardPutBitmap`.

You can discard the bitmap object by passing the ID to the `DiscardBitmap` function.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	scope	integer	Determines which portions of the specified panel are to be copied to the bitmap. See table of valid values below.
	area	Rect	Use this parameter to restrict the area of the panel copied into the bitmap. The rectangle coordinates are relative to the upper-left corner of the panel (directly below the title bar) before the panel is scrolled. Use `VAL_ENTIRE_OBJECT` if you do not want to restrict the area copied.
Output	**bitmapID**	integer	An ID that serves as a handle to the bitmap object.

Return Value

status	integer	Refer to Appendix A for error codes.

Scope

Valid Values	Description
VAL_VISIBLE_AREA	The visible area of the panel is copied to the bitmap, including the frame, menu bar, and scroll bars.
VAL_FULL_PANEL	The entire contents of the panel are copied to the bitmap, excluding the frame, menu bar, and scroll bars. This includes contents that might currently be scrolled off the visible area.

See Also

ClipboardPutBitmap, GetBitmapData, GetCtrlDisplayBitmap, SetCtrlBitmap, PlotBitmap, CanvasDrawBitmap, DiscardBitmap.

GetSystemAttribute

int **status** = **GetSystemAttribute** (int **systemAttribute**, void *__**attributeValue**__);

Purpose

Returns the value of a particular system attribute.

Parameters

Input	**systemAttribute**	integer	ATTR_ALLOW_UNSAFE_TIMER_EVENTS
Output	**attributeValue**	void *	The value of the specified attribute. See Table 3-7 for values associated with this attribute.

Return Value

status	integer	Refer to Appendix A for error codes.

InsertAxisItem

```
int status = InsertAxisItem (int panelHandle, int controlID, int axis,
                             int itemIndex, char itemLabel[], double itemValue);
```

Purpose

This function inserts a string/value pair into the list of label strings associated with a graph or strip chart axis. These strings appear instead of the numerical labels. They appear at the location of their associated values on the graph or strip chart.

To see string labels on an X axis, you must set the ATTR_XUSE_LABEL_STRINGS attribute to TRUE. To see string labels on a Y axis, you must set the ATTR_YUSE_LABEL_STRINGS attribute to TRUE.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the LoadPanel, NewPanel, or DuplicatePanel function.
	controlID	integer	The defined constant (located in the .uir header file) which was assigned to the control in the User Interface Editor, or the ID returned by the NewCtrl or DuplicateCtrl function.
	axis	integer	Specifies the axis for which to insert the string/value pair. Valid values: VAL_XAXIS VAL_LEFT_YAXIS VAL_RIGHT_YAXIS (graphs only)
	itemIndex	integer	The zero-based index into the list at which the item is to be stored. Pass -1 to store the item at the end of the list. See discussion below.
	itemLabel	string	The label string being inserted. The label appears on the axis at the location of the associated value. A maximum of 31 characters are shown.
	itemValue	double	The value to be associated with the label string being inserted. The label string appears on the axis at the location of the value.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

itemIndex does not determine the order in which the labels appear on the axis. It merely represents the order in which LabWindows/CVI stores the string/value pairs. You can use the index as a handle for replacing or deleting label/value pairs.

If you pass -1 for **itemIndex**, the string/value pair is stored at the end of the list.

See Also

`ReplaceAxisItem, DeleteAxisItem, ClearAxisItems, GetNumAxisItems`

LoadMenuBarEx

`int` **menuBarHandle = LoadMenuBarEx** (`int` **destinationPanelHandle**, **filename**`[]`,
`int` **menuBarResourceID**,
`void *`**callingModuleHandle**);

Purpose

`LoadMenuBarEx` loads a menu bar into memory from a User Interface Resource (`*.uir`) that was file created in the User Interface Editor. `LoadMenuBarEx` is similar to `LoadMenuBar`, except that, when you use `LoadMenuBarEx` on Windows 95 and NT, the callback functions you reference in your `.uir` file can be defined in the DLL that contains the call to `LoadMenuBarEx`. On platforms other than Windows 95 and NT, `LoadMenuBarEx` works exactly like `LoadMenuBar`.

Parameters

Input	**destinationPanelHandle**	integer	The handle of the panel on which the menu bar is to reside.
	filename	string	The name of the User Interface Resource File that contains the menu bar.
	panelResourceID	integer	The defined constant assigned to the menu bar in the User Interface Editor.
	callingModuleHandle	void pointer	Usually, the module handle of the calling DLL. You can use `__CVIUserHInst.` Zero indicates the project or executable`.`

Return Value

menuBarHandle	integer	The value you can use in subsequent function calls to specify this menu bar. Negative values indicate that an error occurred. Refer to Appendix A for error codes.

Using this Function

Refer to the function help for `LoadMenuBar` for detailed information on that function. When you call `LoadMenuBar`, the User Interface Library attempts to find the callback functions referenced in the `.uir` file. It searches the symbols defined in the project or in object, library, or DLL import library modules that have been loaded using `LoadExternalModule`. It does not search symbols that are defined in a DLL but not exported in the DLL import library.

You may want a DLL to load a menu bar and link to callback functions defined in (but not exported from) the DLL. You can do this by calling `LoadMenuBarEx`. You must specify the module handle of the DLL in the **callingModuleHandle** parameter. You can do this by using the pre-defined variable `__CVIUserHInst`. (If you pass zero for the **callingModuleHandle**, the function behaves identically to `LoadMenuBar` .)

`LoadMenuBarEx` first searches the DLL symbols to find the callback functions referenced in the `.uir`. If there are any callback functions that it cannot find, it then searches for them in the same manner as `LoadMenuBar`.

`LoadMenuBarEx` expects the DLL to contain a table of the callback functions referenced by the `.uir` files loaded by the DLL. 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 using the **External Compiler Support** command in the **Build** menu of the Project window. You must have a LabWindows/CVI project that lists all of the `.uir` files loaded by the DLL. In the External Compiler Support dialog box, specify the name of an object file to contain the table of callback function names. Then click on the **Create** button to create the object file. You must include the object file in the external compiler project you use to create the DLL.

The External Compiler Support information is contained in the LabWindows/CVI project file. If that project file is loaded and you modify and save any of the `.uir` files, LabWindows/CVI automatically regenerates the object file.

See also

`LoadMenuBar, LoadPanelEx`

LoadPanelEx

```
int panelHandle = LoadPanelEx (int parentPanelHandle, char filename[],
                               int panelResourceID, void *callingModuleHandle);
```

Purpose

LoadPanelEx loads a panel into memory from a User Interface Resource (.uir) file created in the User Interface Editor. LoadMenuBarEx is similar to LoadMenuBar, except that, when you use LoadMenuBarEx your program on Windows 95 and NT, the callback functions you reference in your .uir file can be defined in the DLL that contains the call to LoadPanelEx. On platforms other than Windows 95 and NT, LoadPanelEx works exactly like LoadPanel.

Parameters

Input	**parentPanelHandle**	integer	The handle of the panel into which the panel is loaded as a child panel . Pass 0 to load the panel as a top-level window.
	filename	string	The name of the User Interface Resource File that contains the panel.
	panelResourceID	integer	The defined constant assigned to the panel in the User Interface Editor.
	callingModuleHandle	void pointer	Usually, the module handle of the calling DLL. You can use __CVIUserHInst. Zero indicates the project or executable.

Return Value

panelHandle	integer	The value you can use in subsequent function calls to specify this panel. Negative values indicate that an error occurred. Refer to Appendix A for error codes.

Using this Function

Refer to the function help for LoadPanel for detailed information on that function.

When you call LoadPanel, the User Interface Library attempts to find the callback functions referenced in the .uir file. It searches the symbols defined in the project or in object, library, or DLL import library modules that have been loaded using LoadExternalModule. It does not search symbols that are defined in a DLL but not exported in the DLL import library.

You may want a DLL to load a panel and link to callback functions defined in (but not exported from) the DLL. You can do this by calling LoadPanelEx. You must specify the module

handle of the DLL in the **callingModuleHandle** parameter. You can do this by using the pre-defined variable __CVIUserHInst. (If you pass zero for the **callingModuleHandle**, the function behaves identically to LoadPanel.)

LoadPanelEx first searches the DLL symbols to find the callback functions referenced in the .uir. If there are any callback functions that it cannot find, it then searches for them in the same manner as LoadPanel.

LoadPanelEx expects the DLL to contain a table of the callback functions referenced by the .uir files loaded by the DLL. 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 using the **External Compiler Support** command in the **Build** menu of the Project window. You must have a LabWindows/CVI project that lists all of the .uir files loaded by the DLL. In the External Compiler Support dialog box, specify the name of an object file to contain the table of callback function names. Then click on the **Create** button to create the object file. You must include the object file in the external compiler project you use to create the DLL.

The External Compiler Support information is contained in the LabWindows/CVI project file. If that project file is loaded and you modify and save any of the .uir files, LabWindows/CVI automatically regenerates the object file.

See also

LoadPanel, LoadMenuBarEx

MakePoint

Point **point** = **MakePoint** (int **xCoordinate**, int **yCoordinate**);

Purpose

Returns a Point structure with the specified values. The Point structure defines the location of a point.

This function is useful when calling canvas control functions that require Point structures as input parameters. You can embed a call to MakePoint in calls to these functions, thereby eliminating the need to declare a Point variable.

Parameters

Input	**xCoordinate**	integer	The horizontal location of the point.
	yCoordinate	integer	The vertical location of the point.

Return Value

point	Point	A Point structure containing the specified coordinate values.

See Also

`PointSet, MakeRect`

MakeRect

`Rect` **rect** = **MakeRect** (`int` **top**, `int` **left**, `int` **height**, `int` **width**);

Purpose

Returns a `Rect` structure with the specified values. The `Rect` structure defines the location and size of a rectangle.

This function is useful when calling canvas control functions that require `Rect` structures as input parameters. You can embed a call to `MakeRect` in calls to these functions, thereby eliminating the need to declare a `Rect` variable.

Parameters

Input	**top**	integer	The location of the top edge of the rectangle.
	left	integer	The location of the left edge of the rectangle.
	height	integer	The height of the rectangle.
	width	integer	The width of the rectangle.

Return Value

rect	Rect	A `Rect` structure containing the specified coordinate values.

See Also

`RectSet, MakePoint`

NewBitmap

```
int status = NewBitmap (int bytesPerRow, int pixelDepth, int width, int height,
                        int colorTable[], unsigned char bits[],
                        unsigned char mask[], int *bitmapID);
```

Purpose

Creates a bitmap object. The bitmap ID can then be passed any function that accepts a bitmap, such as `CanvasDrawBitmap` or `ClipboardPutBitmap`.

You can discard the bitmap object by passing its ID to `DiscardBitmap`.

Parameters

Input	**bytesPerRow**	integer	The number of bytes on each scan line of the image.
	pixelDepth	integer	The number of bits per pixel.
	width	integer	The width of the image, in pixels.
	height	integer	The height of the image, in pixels.
	colorTable	integer array	An array of RGB color values, or NULL if **pixelDepth** is greater than 8.
	bits	unsigned char array	An array of bits that determine the colors to be displayed on each pixel in the image.
	mask	unsigned char array	An array containing one bit per pixel in the image. Each bit specifies whether the pixel is actually drawn. May be NULL.
Output	**bitmapID**	integer	An ID that serves as a handle to the bitmap object.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Depending on the **pixelDepth** and **width**, the number of bits per line in the **bits** array might not be an even multiple of 8. If not, then the extra bits needed to get to the next byte boundary are considered "padding". If you set **bytesPerRow** to be a positive number, then the bits for each scan line must start on a byte boundary, and so padding may be required. In fact, you can set **bytesPerRow** to be larger than the minimum number of bytes actually needed. The extra bytes are also considered padding. If you pass -1, there is no padding at all. The bits for each scan line immediately follow the bits for the previous scan line.

The valid values for **pixelDepth** are 1, 4, 8, and 24.

If the **pixelDepth** is 8 or less, the number of entries in the **colorTable** array must equal 2 raised to the power of the **pixelDepth** parameter. The **bits** array contain indices into the **colorTable** array.

If the **pixelDepth** is greater than 8, the **colorTable** parameter is not used. Instead the **bits** array contains actual RGB color values, rather than indices into the **colorTable** array. For a **pixelDepth** of 24, each pixel is represented by a 3-byte RGB value of the form `0xRRGGBB`, where `RR`, `GG`, and `BB` represent the red, green and blue intensity of the color. The `RR` byte should always be at the lowest memory address of the three bytes.

In the **mask** array, a bit value of 1 indicates that the pixel is drawn. 0 indicates that the pixel is not drawn. Exception: If an image has a **pixelDepth** of 1, pixels with a **bits** value of 1 (foreground pixels) are always drawn and the **mask** affects only the pixels with a bitmap of 0 (background pixels). Each row of the mask must be padded to the nearest even-byte boundary. For example, if the width of the image is 21 pixels, then there must be 32 bits (in other words, 4 bytes) of data in each row of the mask.

A mask is useful for achieving transparency.

You may pass NULL if you do not need a mask.

Note: ***Only images that are 32 pixels wide and 32 pixels high can use a mask.***

See Also

`GetBitmapFromFile, GetCtrlBitmap, GetCtrlDisplayBitmap, GetPanelDisplayBitmap, ClipboardGetBitmap, ClipboardPutBitmap, GetBitmapData, SetCtrlBitmap, PlotBitmap, CanvasDrawBitmap, DiscardBitmap.`

PlotScaledIntensity

```
int status = PlotScaledIntensity (int panelHandle, int controlID, void *zArray,
                                  int numberOfXPoints, int numberOfYPoints,
                                  int zDataType, double yGain, double yOffset,
                                  double xGain, double xOffset,
                                  ColorMapEntry colorMapArray[], int hiColor,
                                  int numberofColors, int interpColors,
                                  int interpPixels);
```

Purpose

This function draws a solid rectangular plot in a graph control. It is the same as `PlotIntensity`, except that you can apply scaling factors and offsets to the data values.

The plot consists of pixels whose colors correspond to the magnitude of data values in a two-dimensional array and whose coordinates correspond to the locations of the same data values in the array, scaled by **xGain** and **yGain** and offset by **xOffset** and **yOffset**. For instance the pixel associated with **zArray**[2][3] is located at the following coordinates.

{x = 3***xGain** + **xOffset**, y = 2***yGain** + **yOffset**}.

The lower left corner of the plot area is located at

{**xOffset**, **yOffset**}.

The upper right corner of the plot area is located at

{(X-1)***xGain** + **xOffset**, (Y-1)***yGain** + **yOffset**},

where

X = Number of X Points
Y = Number of Y Points

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	zArray	numeric array	An array that contains the data values that are to be converted to colors.
	numberOfXPoints	integer	The number of points to be displayed along the x-axis in each row.
	numberOfYPoints	integer	The number of points to be displayed along the y-axis in each column.
	zDataType	integer	Specifies the data type of the elements in **zArray**, as well as the data type of the Color Map values.
	yGain	double	Specifies the scaling factor to be applied to the vertical locations implied by **zArray** vertical index values.
	yOffset	double	Specifies the offset to be added to the vertical locations implied by **zArray** vertical index values.

continues

Parameters (Continued)

	xGain	double	Specifies the scaling factor to be applied to the horizontal locations implied by **zArray** horizontal index values.
	xOffset	double	Specifies the offset to be added to the horizontal locations implied by **zArray** horizontal index values.
	colorMapArray	ColorMapEntry	An array of `ColorMapEntry` structures which specifies how the data values in **zArray** are translated. The maximum number of entries is 255.
	hiColor	integer	The RGB value to which all **zArray** values that are higher than the highest data value in **colorMapArray** are translated.
	numberOfColors	integer	The number of entries in **colorMapArray**. Must be less than or equal to 255.
	interpColors	integer	Indicates how to assign colors to **zArray** data values that do not exactly match the data values in the **colorMapArray**.
	interpPixels	integer	Indicates how pixels between the pixels assigned to the **zArray** values are colored.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

zArray must be one of the following data types (specified in **zDataType**).

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

The locations at which the colors are shown on the graph depend on that location of the data values in **zArray**.

- **zArray** should be a two-dimensional array of the following form.

  ```
  zArray[numberOfYPoints][numberOfXPoints]
  ```

- Each element of the array is associated with a pixel on the graph. The pixel associated with element **zArray**[y][x] is located at {x * **xGain** + **xOffset**, y * **yGain** + **yOffset**} on the graph.

colorMapArray contains up to 255 `ColorMapEntry` structures, which consist of:

```
union {
   char valChar;
   int valInt;
   short valShort;
   float valFloat;
   double valDouble;
   unsigned char valUChar;
   unsigned long valULong;
   unsigned short valUShort;
} data Value;
int color;   /* RGB value */
```

The Color Map array defines how data values in **zArray** are translated into color values. If a data value matches exactly to a data value in one of the `ColorMapEntry` structures, then it is converted to the corresponding color. Otherwise, the following conditions apply.

- If **interpColors** is zero, the color associated with the next higher data value is used.
- If **interpColors** is nonzero, the color is computed using a weighted mean of the colors associated with the Color Map data values immediately above and below the **zArray** value.
- Regardless of the value of **interpColors**, the following conditions apply.
  - Data below the lowest Color Map data value are assigned the color of the lowest Color Map data value.
  - Data values above the highest Color Map data value are assigned the value of the hiColor parameter.

If **interpColors** is nonzero, the **numberOfColors** must be greater than or equal to 2.

The **colorMapArray** entries do not need to be in sorted order.

interpPixels indicates how pixels between the pixels assigned to the **zArray** values are colored. If **interpPixels** is zero, an unassigned pixel is given the same color as the closest assigned pixel. If **interpPixels** is nonzero, an unassigned pixel is first given a data value using a weighted mean of the data values associated with the four closest assigned pixels. Then the color is calculated using the Color Map.

Performance Considerations

If **interpPixels** is zero, the performance depends on the number of data points in **zArray**.

If **interpPixels** is nonzero, the performance depends on the total number of pixels in the plot area.

PointEqual

`int` **pointsAreEqual** = **PointEqual** (`Point` **point1**, `Point` **point2**);

Purpose

Indicates if two points are the same.

Returns 1 if the x and y values of two specified points are the same. Returns 0 otherwise.

Parameters

Input	**point1**	Point	A point structure.
	point2	Point	A point structure.

Return Value

pointsAreEqual	integer	An indication of the two points are the same.

Return Codes

1	The x and y coordinates in the two Point structures are the same.
0	The x and y coordinates in the two Point structures are not the same.

See Also

`MakePoint`

PointPinnedToRect

`void` **PointPinnedToRect** (`Point` **point**, `Rect` **rect**, `Point *`**pinnedPoint**);

Purpose

This function ensures that a point is within a specified rectangular area. If the point is already enclosed by the rectangle, the location of the point remains unchanged. If the point is outside the rectangle, its location is set to the nearest point on the edge of the rectangle.

The `Point` structure containing the original location is not modified. The calculated location is stored in another Point structure.

Parameters

Input	**point**	Point	A Point structure specifying the original location of the point.
	rect	Rect	A `Rect` structure specifying the rectangle to which the point is to be pinned.
Output	**pinnedPoint**	Point	The `Point` structure in which the calculated location is stored.

Return Value

None

See Also

`MakePoint, MakeRect.`

PointSet

`void` **PointSet** (`Point *`**point**, `int` **xCoordinate**, `int` **yCoordinate**);

Purpose

Sets the values in an existing Point structure. The `Point` structure defines the location of a point.

Parameters

Input	**xCoordinate**	integer	The new horizontal location of the point.
	yCoordinate	integer	The new vertical location of the point.
Output	**point**	Point	The `Point` structure in which the new values are set.

Return Value

None

See Also

`MakePoint`

RectBottom

`int` **bottom** = **RectBottom** (`Rect` **rect**);

Purpose

Returns the y coordinate of the bottom edge a rectangle. The bottom edge is **not** enclosed by the rectangle. It is computed as follows.

bottom = **rect.top** + **rect.height**

Parameter

Input	**rect**	Rect	Specifies the location and size of a rectangle.

Return Value

bottom	integer	The y coordinate of the bottom of the rectangle. The bottom is not enclosed by the rectangle, and is equal to **rect.top** + **rect.height**.

See Also

`RectRight`

RectCenter

`void` **RectCenter** (`Rect` **rect**, `Point *`**center**);

Purpose

Calculates the location of the center point of the specified rectangle. For even heights (or widths), the center point is rounded towards the top (or left).

Parameters

Input	**rect**	Rect	Specifies the location and size of a rectangle.
Output	**center**	Point	Specifies the location of the center of the rectangle.

Return Value

None

RectContainsPoint

`int` **containsPoint** = **RectContainsPoint** (`Rect` **rect**, `Point` **point**);

Purpose

Returns 1 if the specified rectangle encloses the specified point. Returns 0 otherwise. The rectangle is considered to enclose the point if the point is in the interior of the rectangle or on its frame.

Parameters

Input	**rect**	Rect	Specifies the location and size of a rectangle.
Output	**point**	Point	Specifies the location of the center of the rectangle.

Return Value

containsPoint	integer	Indicates if **rect** contains **point**.

Return Codes

1	**point** is in the interior or on the frame of the rectangle specified by **rect**.
0	**point** is outside the frame of the rectangle specified by **rect.**

RectContainsRect

`int` **containsRect** = **RectContainsRect** (`Rect` **rect1**, `Rect` **rect2**);

Purpose

Returns 1 if the first rectangle encloses the second rectangle. Returns 0 otherwise. A rectangle is considered to enclose another rectangle if every point of the second rectangle is in the interior or on the frame of the first rectangle. (A rectangle encloses itself.)

Parameters

Input	**rect1**	Rect	Specifies the location and size of a rectangle.
	rect2	Rect	Specifies the location and size of a rectangle.

Return Value

containsRect	integer	Indicates if **rect1** encloses **rect2**.

Return Codes

1	**rect1** encloses **rect2.**
0	**rect1** does not enclose **rect2.**

RectEmpty

`int` **isEmpty** = **RectEmpty** (`Rect` **rect**);

Purpose

Returns 1 if the specified rectangle is empty. Returns 0 otherwise. A rectangle is considered to be empty if either its height or width is less than or equal to zero.

Parameter

Input	**rect**	Rect	Specifies the location and size of a rectangle. A `Rect` structure.

Return Value

isEmpty	integer	Indicates if the rectangle specified by **rect** is empty.

Return Codes

1	Either **rect.height** or **rect.width** of is less than or equal to zero.
0	Both **rect.height** and **rect.width** are greater than zero.

RectEqual

`int` **areEqual** = **RectEqual** (`Rect` **rect1**, `Rect` **rect2**);

Purpose

Returns 1 if the location and size of the two specified rectangles are identical. Returns 0 otherwise.

Parameters

Input	**rect1**	Rect	Specifies the location and size of a rectangle.
	rect2	Rect	Specifies the location and size of a rectangle.

Return Value

areEqual	integer	Indicates if the top, left, height, and width values in **rect1** are identical to those of **rect2**.

Return Codes

1	**rect1** and **rect2** have the identical top, left, height, and width values.
0	**rect1** and **rect2** do not have the identical top, left, height, and width values.

RectGrow

`void` **RectGrow** (`Rect` *__rect__, `int` **dx**, `int` **dy**);

Purpose

Modifies the values in a `Rect` structure so that the rectangle it defines grows or shrinks around its current center point.

Parameters

Input/Output	**rect**	Rect	On input, specifies the size and location of a rectangle. On output, specifies a rectangle of a different size but the same center point.
Input	**dx**	integer	The amount to grow the rectangle horizontally. Use a negative value to shrink the rectangle horizontally.
	dy	integer	The amount to grow the rectangle vertically. Use a negative value to shrink the rectangle vertically.

Return Value

None

RectIntersection

`int` **rectsIntersect** = **RectIntersection** (`Rect` **rect1**, `Rect` **rect2**, `Rect *`**intersectionRect**);

Purpose

Returns an indication of whether two rectangles are intersecting. If they are, the function fills in a `Rect` structure describing the intersection area.

Parameters

Input	**rect1**	Rect	Specifies the location and size of a rectangle.
	rect2	Rect	Specifies the location and size of a rectangle.
Output	**intersectionRect**	Rect	The `Rect` structure set to the largest rectangle enclosed by both **rect1** and **rect2**. If **rect1** and **rect2** do not intersect, this parameter is set to an empty rectangle (height and width of zero). You may pass 0 for this parameter.

Return Value

rectsIntersect	integer	Indicates if **rect1** and **rect2** intersect (have any points in common).

Return Codes

1	**rect1** and **rect2** intersect.
0	**rect1** and **rect2** do not intersect.

RectMove

`void` **RectMove** (`Rect *`**rect**, `Point` **point**);

Purpose

Modifies a `Rect` structure so that the top, left corner of the rectangle it defines is at the specified point.

Parameters

Input/Output	**rect**	Rect	On input, specifies the size and location of a rectangle. On output, specifies a rectangle of the same size but a different location.
Input	**point**	Point	A Point structure specifying the new location of the top, left corner of the rectangle.

Return Value

None

RectOffset

`void` **RectOffset** (`Rect *`**rect**, `int` **dx**, `int` **dy**);

Purpose

Modifies the values in a `Rect` structure to shift the location of the rectangle it defines.

Parameters

Input/Output	**rect**	Rect	On input, specifies the size and location of a rectangle. On output, specifies a rectangle of the same size but a different location.
Input	**dx**	integer	The amount to shift the rectangle horizontally. Use a positive value to shift the rectangle to the right. Use a negative value to shift the rectangle to the left.
	dy	integer	The amount to shift the rectangle vertically. Use a positive value to shift the rectangle down. Use a negative value to shift the rectangle up.

Return Value

None

RectRight

`int` **rightEdge** = **RectRight** (`Rect` **rect**);

Purpose

Returns the x coordinate of the right edge a rectangle. The right edge is **not** enclosed by the rectangle. It is computed as follows.

rightEdge = **rect.left** + **rect.width**

Parameter

Input	**rect**	Rect	Specifies a rectangle.

Return Value

rightEdge	integer	The x coordinate of the right edge of the rectangle. The right edge is not enclosed by the rectangle, and is equal to **rect.left** + **rect.width**.

See Also

`RectBottom`

RectSameSize

`int` **areSameSize** = **RectSameSize** (`Rect` **rect1**, `Rect` **rect2**);

Purpose

Returns 1 if the two specified rectangle have the same height and width. Returns 0 otherwise.

Parameters

Input	**rect1**	Rect	Specifies the location and size of a rectangle.
	rect2	Rect	Specifies the location and size of a rectangle.

Return Value

areSameSize	integer	Indicates if the height, and width values in **rect1** are identical to those of **rect2**.

Return Codes

1	**rect1** and **rect2** have the identical height, and width.
0	**rect1** and **rect2** do not have the identical height and width.

RectSet

`void` **RectSet** (`Rect *`**rect**, `int` **top**, `int` **left**, `int` **height**, `int` **width**);

Purpose

Sets the values in an existing `Rect` structure. The `Rect` structure defines the location and size of a rectangle.

Parameters

Input	**top**	integer	The new location of the top edge of the rectangle.
	left	integer	The new location of the left edge of the rectangle.
	height	integer	The new height of the rectangle.
	width	integer	The new width of the rectangle.
Output	**rect**	Rect	The `Rect` structure in which the new values are set.

Return Value

None

See Also

`MakeRect`

RectSetBottom

`void` **RectSetBottom** (`Rect *`**rect**, `int` **bottom**);

Purpose

Sets the height of a `Rect` structure so that the bottom edge of the rectangle it defines is at the specified location. The bottom edge of the rectangle is **not** enclosed by the rectangle and is equal to the top plus the height.

Parameters

Input/Output	**rect**	Rect	On input, specifies the size and location of a rectangle. On output, specifies the same rectangle except with a different bottom edge.
Input	**bottom**	integer	The y coordinate of the new bottom edge.

Return Value

None

RectSetCenter

`void` **RectSetCenter** (`Rect *`**rect**, `Point` **center**);

Purpose

Modifies the values of a `Rect` structure so that it retains its current size but is centered around the specified point.

Parameters

Input/Output	**rect**	Rect	On input, specifies the size and location of a rectangle. On output, specifies a rectangle of the same size but with a different center point.
Input	**center**	Point	A Point structure specifying the location of the new center point of the rectangle.

Return Value

None

RectSetFromPoints

`void` **RectSetFromPoints** (`Rect *`**rect**, `Point` **point1**, `Point` **point2**);

Purpose

Sets the values in a `Rect` structure so that it defines the smallest rectangle that encloses two specified points. Each point is located on a corner of the frame of the rectangle.

Parameters

Input	**point1**	Point	Specifies the location of a point.
	point1	Point	Specifies the location of a point.
Output	**rect**	Rect	The `Rect` structure that is set to enclose the specified points.

Return Value

None

RectSetRight

`void` **RectSetRight** (`Rect *`**rect**, `int` **right**);

Purpose

Sets the width of a `Rect` structure so that the right edge of the rectangle it defines is at the specified location. The right edge of the rectangle is **not** enclosed by the rectangle and is equal to the left edge plus the width.

Parameters

Input/Output	**rect**	Rect	On input, specifies the size and location of a rectangle. On output, specifies the same rectangle except with a different right edge.
Input	**right**	integer	The x coordinate of the new right edge.

Return Value

None

RectUnion

`void` **RectUnion** (`Rect` **rect1**, `Rect` **rect2**, `Rect *`**unionRect**);

Purpose

Calculates the smallest rectangle which encloses two specified rectangles.

Parameters

Input	**rect1**	Rect	Specifies the size and location of a rectangle.
	rect1	Rect	Specifies the size and location of a rectangle.
Output	**unionRect**	Rect	The `Rect` structure that is set to the smallest rectangle that encloses both **rect1** and **rect2**.

Return Value

None

ReplaceAxisItem

`int` **status** = **ReplaceAxisItem** (`int` **panelHandle**, `int` **controlID**, `int` **axis**, `int` **itemIndex**, `char` **itemLabel**`[]`, `double` **itemValue**);

Purpose

This function replaces the string/value pair at the specified index in the list of label strings for a graph or strip chart axis. These strings appear in place of the numerical labels. They appear at the location of their associated values on the graph or strip chart.

To see string labels on an X axis, you must set the `ATTR_XUSE_LABEL_STRINGS` attribute to TRUE. To see string labels on a Y axis, you must set the `ATTR_YUSE_LABEL_STRINGS` attribute to TRUE.

The original list of label strings can be created in the User Interface Editor or by calling `InsertAxisItem`.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.

	axis	integer	Specifies the axis for which to replace the string/value pair at the specified index. Valid values:
			`VAL_XAXIS`
			`VAL_LEFT_YAXIS`
			`VAL_RIGHT_YAXIS` (graphs only)

	itemIndex	integer	The zero-based index of the item to be replaced.
	itemLabel	string	The string to replace the existing string in the item at the specified index. If you pass 0, then the existing string is not replaced. A maximum of 31 characters from the string are shown in an axis label.
	itemValue	double	The value to replace the existing value in the string/value pair at the specified index. The string appears as an axis label at the location of the value.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

`InsertAxisItem, DeleteAxisItem, ClearAxisItems, GetNumAxisItems`

SetAxisScalingMode

```
int status = SetAxisScalingMode (int panelHandle, int controlID, int axis,
                                 int axisScaling, double min, double max);
```

Purpose

Sets the scaling mode and the range of any graph axis or the Y axis of a strip chart.

This function is not valid for the X axis of a strip chart. To set the X offset and X increment for a strip chart, use the `SetCtrlAttribute` function with the `ATTR_XAXIS_OFFSET` and `ATTR_XAXIS_GAIN` attributes.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the `LoadPanel`, `NewPanel`, or `DuplicatePanel` function.
	controlID	integer	The defined constant (located in the `.uir` header file) which was assigned to the control in the User Interface Editor, or the ID returned by the `NewCtrl` or `DuplicateCtrl` function.
	axis	integer	Specifies for which axis to set the mode and range. Valid values: `VAL_XAXIS` (graphs only) `VAL_LEFT_YAXIS` (graphs and strip charts) `VAL_RIGHT_YAXIS` (graphs only)
	axisScaling	integer	The scaling mode to be used for the axis. See table below.
	min	double	The minimum axis value when the axis is configured for manual scaling.
	max	double	The maximum axis value when the axis is configured for manual scaling.

Return Value

status	integer	Refer to Appendix A for error codes.

Parameter Discussion

axisScaling must be one of the following values.

Valid Value	Description
VAL_MANUAL	The axis is set to manual scaling, and its range is defined by **min** and **max**.
VAL_AUTOSCALE	The axis is set to auto scaling. **min** and **max** are not used. You cannot use auto scaling in strip charts.
VAL_LOCK	The axis is set to manual scaling using the current (usually auto-scaled) minimum and maximum values on the axis. You cannot use VAL_LOCK in strip charts.

If **axisScaling** is VAL_MANUAL**, max** must exceed **min**.

See Also

GetAxisScalingMode

SetCtrlBitmap

int **status** = **SetCtrlBitmap** (int **panelHandle**, int **controlID**, int **imageID**,
int **bitmapID**);

Purpose

Sets the image of a control from a bitmap object. Can be used to replace an existing image on a control or to create a new image on a control.

The following control types can contain images:

picture controls
picture rings
picture buttons
graph controls

For picture controls, this function can be used as an alternative to DisplayImageFile.

For picture buttons, this function can be used as an alternative to SetCtrlAttribute with the attribute set to ATTR_IMAGE_FILE.

For picture rings, this function can be used as an alternative to ReplaceListItem. (To add a new entry, first call InsertListItem with a NULL value, and then call SetCtrlBitmap.)

For graphs, you must first call PlotBitmap with a NULL filename. Then call SetCtrlBitmap.

If you want to delete an image, call SetCtrlBitmap with 0 as the value for the bitmap ID.

Parameters

Input	**panelHandle**	integer	The specifier for a particular panel that is contained in memory. This handle will have been returned by the LoadPanel, NewPanel, or DuplicatePanel function.
	controlID	integer	The defined constant (located in the .uir header file) which was assigned to the control in the User Interface Editor, or the ID returned by the NewCtrl or DuplicateCtrl function.
	imageID	integer	For picture rings, the zero-based index of an image in the ring. For graphs, this argument is the **plotHandle** returned from PlotBitmap. For picture controls and picture buttons, this argument is ignored.
	bitmapID	integer	The ID of the bitmap object containing the new image. The ID must have been obtained from NewBitmap, GetBitmapFromFile, GetCtrlBitmap, ClipboardGetBitmap, GetCtrlDisplayBitmap, or GetPanelDisplayBitmap.

Return Value

status	integer	Refer to Appendix A for error codes.

See Also

NewBitmap, GetBitmapFromFile, GetCtrlBitmap, GetCtrlDisplayBitmap, GetPanelDisplayBitmap, ClipboardGetBitmap.

SetSystemAttribute

`int` **status** = **SetSystemAttribute** (`int` **systemAttribute**, …);

Purpose

Sets the value of a particular system attribute.

Parameters

Input	**systemAttribute** **attributeValue**	integer depends on the attribute	`ATTR_ALLOW_UNSAFE_TIMER_EVENTS` The value of the specified attribute. See Table 3-7 for values associated with this attribute.

Return Value

status	integer	Refer to Appendix A for error codes.

Chapter 4
Updates to the Standard Libraries Reference Manual

Chapter Contents

Changes to the ANSI C Library and Low-Level I/O Functions

This chapter discusses changes made to the LabWindows/CVI ANSI C library and the low-level I/O functions. The ANSI C library is documented in Chapter 1 of the *Standard Libraries Reference Manual*. The low-level I/O functions are implemented by LabWindows/CVI only under Windows. There is no printed documentation or function panels for the low-level I/O functions. They are declared in `lowlvlio.h`, which is in the `cvi/include` directory. (On the SPARCstation, you can call the low-level I/O functions in the Sun system library.)

errno Set by File I/O Functions

The `errno` global variable is now 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\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.

Note: ***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
```

New Low-Level I/O Function

The `sopen` function has been added to the set of low-level I/O functions. You can use `sopen` to restrict the ability of other applications to open a file while your application has that file open. The documentation for `sopen` is in `lowlvlio.h`.

New ANSI C Library Function

The `fdopen` function has been added to the ANSI C function panels and the `stdio.h` include file.

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)
`OpenFile` (Formatting and I/O Library)

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`, `sopen`, or `OpenFile`.
	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.

Changes to the Formatting and I/O Library

This chapter discusses changes made to the LabWindows/CVI Formatting and I/O Library. The Formatting and I/O Library is documented in the Chapter 2 of the *Standard Libraries Reference Manual*.

Improved File I/O Error Reporting

Two new functions have been added to give you more specific error information when a file I/O error occurs in a Formatting and I/O function.

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.

Changes to the GPIB Library

This chapter discusses changes made to the LabWindows/CVI GPIB Library.

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 3.1

For Windows 3.1, there are no GPIB library changes between LabWindows/CVI versions 3.1 and 4.0. All GPIB boards are supported in the same way under Windows 3.1.

Windows 95

There are two different 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 (Plug and Play)
- 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 64k byte 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

In LabWindows/CVI version 3.1, the `ibInstallCallback` function was added. You 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, the conditions under which the function can be called have been expanded. You can specify the callback 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

The ability to install *asynchronous* callbacks have been added for Windows NT and for 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`.

New Functions

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 be 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 do 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`

ThreadIbcnt

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

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`);

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`);

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.

continues

Return Codes (Continued)

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);

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

Changes to the RS-232 Library

This chapter discusses changes made to the LabWindows/CVI RS-232 Library. The RS-232 library is documented in Chapter 8 of the *Standard Libraries Reference Manual.*

New Function

The following function has been added to the RS-232 Library.

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 in the *LabWindows/CVI Standard Libraries Reference Manual*.

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.

Changes to the Utility Library

This chapter discusses changes made to the LabWindows/CVI Utility Library. The Utility Library is documented in Chapter 8 of the *Standard Libraries Reference Manual.*

Corrections to Documentation

The following corrections should be made to Chapter 8, *Utility Library*, of the *Standard Libraries Reference Manual.*

LaunchExecutableEx

In the *Launching LabWindows/CVI Runtime Executables* section, the following line,

```
c:\cvi\cvirt3.exe c:\test\myapp.exe c:\test\myargs
```

should be replaced by the following.

```
c:\cvi\cvirt40.exe  c:\test\myappe.exe  myargs
```

Immediately following that line, add the following text.

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
```

Modifications to Existing Functions for Windows 95 and NT

The following modifications have been made to existing functions.

DisableTaskSwitching

For Windows 95, the task list is replaced by the task bar. If you follow the instructions for forcing your window to cover the entire screen, the task bar is also covered and cannot be accessed by the end-user.

For Windows NT, `DisableTaskSwitching` has no effect. To achieve the same results, you must disable the Task Manager and arrange for your LabWindows/CVI application to be brought up in place of the Program Manager. You can do this by making following changes to the registry entry on the following key.

```
HKEY_LOCAL_MACHINE\Software\Microsoft\CurrentVersion\Winlogon
```

- Change the value for `SHELL` to the pathname of your application executable.
- Add a value with the name of `TASKMAN`. Set the data to an empty string.

Certain user operations on MS Windows, such as dragging a window or pulling down the system menu, interfere with the processing of events. The most complete way to keep such operations from interfering with real-time processing is to fix the position of your panels, hide the system menu, and so on. Now, as an alternative, you can enable timer callbacks on Windows 95 and NT during some, but not all, of these operations. You do this with the following function call.

```
SetSystemAttribute (ATTR_ALLOW_UNSAFE_TIMER_EVENTS, 1);
```

(This can be unsafe. See the discussion of the `ATTR_ALLOW_UNSAFE_TIMER_EVENTS` attribute in Chapter 3 of this document.)

LoadExternalModule

Add the following two return codes.

-25	DLL initialization function failed.
-26	module already loaded with different calling module handle. (See `LoadExternalModuleEx`.)

In the second paragraph of the *Parameter Discussion* section, change

In Windows

to

In Windows 3.1

After the second paragraph of the *Parameter Discussion* section, add the following.

In 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 the *Using This Function* section, add the following before the example.

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.

SetSystemDate and SetSystemTime

On Windows NT, you must have system administrator status to use the `SetSystemDate` and `SetSystemTime` functions.

EnableInterrupts and DisableInterrupts

On Windows NT, the `EnableInterrupts` and `DisableInterrupts` functions have no effect.

Revised Error Codes

New error codes have been added for some of the Utility library functions. In some cases, the existing error codes have been reduced in scope or changed in value. The following is the complete set of error codes for each of the functions for which error codes have been added.

CopyFile

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.

DeleteDir

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

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.

GetDir

0	Success.
-3	General I/O error occurred.
-4	Insufficient memory to complete operation.

GetDrive

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.

GetFileDate

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.

GetFileSize

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.

GetFileTime

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.

GetFirstFile

0	Success.
-1	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.

MakeDir

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.

RenameFile

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.

SetFileDate

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.

SetFileTime

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.

New Functions

The following functions have been added to the Utility library.

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.

GetBreakOnProtectionErrors

`int` **state** = **GetBreakOnProtectionErrors** (`void`);

Purpose

This function returns the state of the **break on protection errors** option. 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 option, 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

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).

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.

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 if the memory actually exists. If it does, success is returned but no data is read.

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.

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);
```

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 if the memory actually exists. If it does, success is returned but no data is written.

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**.

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.

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 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 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 4-1. Easy I/O for DAQ Function Tree

Easy I/O for DAQ	
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*

continues

Table 4-1. Easy I/O for DAQ Function Tree (Continued)

Analog Output	
AO Update Channel	*AOUpdateChannel*
AO Update Channels	*AOUpdateChannels*
AO Generate Waveform(s)	*AOGenerateWaveforms*
AO Clear Waveform(s)	*AOClearWaveforms*
Digital Input/Output	
Read From Digital Line	*ReadFromDigitalLine*
Read From Digital Port	*ReadFromDigitalPort*
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*

- 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 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 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 4-2.

Table 4-2. 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

The 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 4-5.

Parameter Discussion

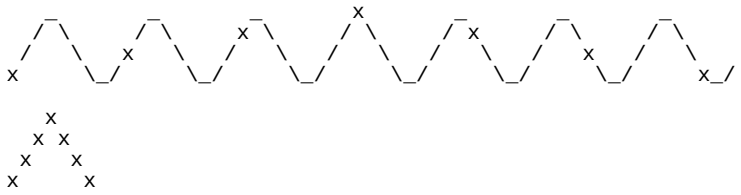
channelString is the analog input channels that are to be sampled. Refer to the *Channel String for Analog Input Functions* subsection of the *Easy I/O for DAQ Library Function Overview* section of this chapter for the syntax of this string.

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.

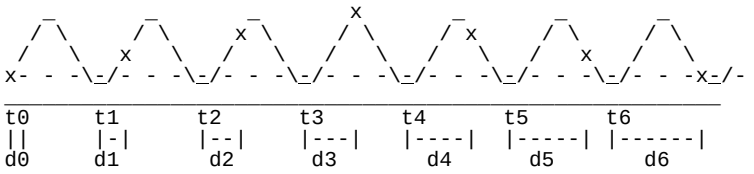
In the following figure:

`tn` => the nth trigger condition

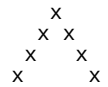
`dn` => delay between the nth trigger and the nth conversion

`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 are 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 4-5.

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 are 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 4-5.

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 4-5.

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 4-5.

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 Scans to Read. 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 4-5.

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 4-5.

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**.

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 4-5.

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 the *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 4-5.

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.

continues

Parameters (Continued)

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 4-5.

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 4-5.

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 4-5.

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 4-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 4-2, 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 4-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 4-2, 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 are cascaded with the specified counter. Counter+1 is defined as shown in Table 4-3.

Table 4-3. 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 4-4.

Table 4-4. 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 4-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 4-2, 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 are cascaded with the specified counter. **counter**+1 is defined as shown in Table 4-3 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 4-4.

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 4-5.

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 4-5.

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 4-5.

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.

continues

Parameters (Continued)

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 4-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 4-2, 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.

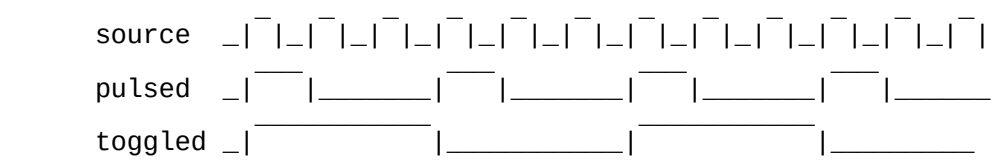
Set the **timebaseDivisor** to the desired value. For a value of `N` and a pulsed output, an output pulse equal to the period of the source or timebase signal appears on counter's OUT pin once each `N` cycles of that signal. For a toggled output, the output toggles after each `N` cycles. The toggled 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 4-5.

Parameter Discussion

counter is the counter to be used for the counting operation. The valid counters are shown in Table 4-2, 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.

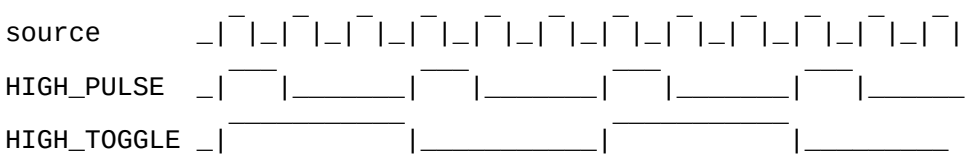
For a Timebase Divisor of N and a pulsed output, an output pulse equal to the period of the source or timebase signal appears on counter's OUT pin once each N cycles of that signal For a toggled output, the output toggles after each N cycles. The toggled output frequency is thus half that of the pulsed 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 4-5.

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 4-5.

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 4-5.

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 4-5.

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" vs. "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 4-5.

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 4-5.

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 4-5.

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 4-5.

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 4-5.

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 4-5.

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.

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 4-5.

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 4-5.

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 4-5. 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.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-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.
-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.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-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.
-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.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-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.
-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	**badScanListError** 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.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-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.
-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.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-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.
-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	**invalidDSPhandleError** 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.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-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.
-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.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-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	**invalidRegenModeError** This device does not support the specified analog output regeneration mode.
-10680	**badChanGainErr** All channels must have an identical setting for this device.
-10681	**badChanRangeErr** All channels of this device must have the same range.
-10682	**badChanPolarityErr** All channels of this device must have the same polarity.
-10683	**badChanCouplingErr** All channels of this device must have the same coupling.
-10684	**badChanInputModeErr** All channels of this device must have the same input range.
-10685	**clkExceedsBrdsMaxConvRate** 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	**extScanClockError** Cannot use an external scan clock when performing a single scan of a single channel.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-10694	**unsafeSamplingFreqError** 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.
-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	**gateSignalError** Attempted to start a pulse width measurement with the pulse in the active state.

continues

Table 4-5. Easy I/O for DAQ Error Codes (Continued)

-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.
-10848	**noDMACountAvailErr** The driver could not obtain a valid reading from the transfer-count register in the DMA controller.
-10849	**openFileError** Unable to open a file.
-10850	**closeFileError** Unable to close a file.
-10851	**fileSeekError** Unable to seek within a file.
-10852	**readFileError** Unable to read from a file.
-10853	**writeFileError** Unable to write to a file.
-10854	**miscFileError** An error occurred accessing a file.
-10880	**updateRateChangeError** 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	**gpctrDataLossError** One or more data points may have been lost during buffered GPCTR operations due to speed limitations of your system.

Chapter 5
General Updates to LabWindows/CVI

Chapter Contents

Configuring LabWindows/CVI in Windows 95 and NT

Configuring LabWindows/CVI in Windows 95 and NT is very similar to configuring it under Windows 3.1. Configuration is discussed in Chapter 1, C*onfiguring LabWindows/CVI* of the *User Manual*. This chapter discusses the changes to the standard method for configuration under Windows.

How To Set Configuration Options

In Windows 3.1, the configuration options that are needed to launch the LabWindows/CVI development environment are stored in the `win.ini` file. In Windows 95 and NT, these are stored in the Registry under the following key.

```
HKEY_LOCAL_MACHINE\Software\National Instruments\CVI
```

There are also configuration settings for the run-time library DLLs. See the *Configuring the Run-Time Engine* section in Chapter 4, *Creating and Distributing Standalone Executables*, of the *Programmer Reference Manual* for a description of the settings. In Windows 95 and NT, these settings are stored in the Registry under the following key.

```
HKEY_LOCAL_MACHINE\Software\National Instruments\CVI Run-time Engine
```

Option Descriptions

Directory Options

cfgdir

The `cfgdir` option is not used for Windows 95 and NT. In Windows 3.1, the `cfgdir` option identifies the directory for the LabWindows/CVI configuration file, `cvi.ini`, which stores the current state of the options that you can set within the LabWindows/CVI development environment.

In Windows 95 and NT, `cvi.ini` is not used. Instead, the current state of the options that you can set within the environment are stored in the Registry under the following key.

```
HKEY_CURRENT_USER\Software\National Instruments\CVI
```

Changes to the Data Acquisition Library

This chapter discusses changes made to the LabWindows/CVI Data Acquistion Library.

Event Function Parameter Data Types Changed for Windows 95 and NT

Some parameters in the Data Acquisition event handling functions that are two bytes under Windows 3.1 have been increased in size to four bytes under Windows 95 and NT. Typedefs have been added to the include file (`dataacq.h`) and the function panels so that you can write source code that works on all three platforms.

The following table shows the typedefs and the intrinsic types under for the different platforms.

Table 5-1. Typedefs and Intrinsic Types for Different Platforms

Typedef	Windows 3.1	Windows 95/NT
`DAQEventHandle`	`short`	`int`
`DAQEventMsg`	`short`	`int`
`DAQEventWParam`	`unsigned short`	`unsigned int`
`DAQEventLParam`	`unsigned long`	`unsigned long`

The following function prototypes have been affected by this change.

```
typedef void (*DAQEventCallbackPtr) (DAQEventHandle handle, DAQEventMsg
        msg, DAQEventWParam wParam, DAQEventLParam lParam);

short Config_Alarm_Deadband (short board, short mode, char channelString[],
                             double triggerLevel, deadbandWidth, DAQEventHandle
                             handle,
                             DAQEventMsg alarmOnMessage,
                             DAQEventMsg alarmOffMessage,
                             DAQEventCallbackPtr EventFunction);

short Config_ATrig_Event_Message (short board, short mode, char channelString[],
                                  double triggerLevel, double windowSize, short
                                  triggerSlope,
                                  long triggerSkipCount, unsigned long preTriggerScans,
                                  unsigned long postTriggerScans, DAQEventHandle
                                  handle, DAQEventMsg message,
                                  DAQEventCallbackPtr eventFunction);
```

```
short Config_DAQ_Event_Message (short board, short mode, char channelString[],
                                short DAQEvent, unsigned long triggerValue0,
                                unsigned long triggerValue1, long triggerSkipCount,
                                unsigned long preTriggerScans,
                                unsigned long postTriggerScans, DAQEventHandle
                                handle, DAQEventMsg message, DAQEventCallbackPtr
                                eventFunction);

short Get_DAQ_Event (unsigned long timeOut, DAQEventHandle *handle,
                     DAQEventMsg *message, DAQEventWParam *wParam,
                     DAQEventLParam *lParam);

short Peek_DAQ_Event (unsigned long timeOut, DAQEventHandle *handle,
                      DAQEventMsg *message, DAQEventWParam *wParam,
                      DAQEventLParam *lParam);
```

Source Code Changes Needed

If you have written source code for Windows 3.1 that uses these functions and you want to use the source code under Windows 95 or NT, you must modify your source code.

You must change the parameter declarations for all of your event callback functions to match the new callback function prototype. You must also use the new typedefs in the declarations of variables that are passed by reference to `Get_DAQ_Event` and `Peek_DAQ_Event`.

Differences in Current NI-DAQ® API for Windows NT

The following functions are not in the current NI-DAQ API for Windows NT. They will be added in a new release of the NI-DAQ DLL for Windows NT in the latter part of 1996.

```
SC_2040_Config
Calibrate_E_Series
Calibrate_1200
AI_Read_Scan
AI_Vread_Scan
AO_Change_Parameter
SCXI_Config_Filter
```

The original NI-DAQ API for Windows NT is different than the NI-DAQ API for Windows 95. Some parameters that are 2-byte integers in the Windows 95 API are 4-byte integers in the original Windows NT API. In the latter part of 1996, the NI-DAQ API for Windows NT will be changed to be consistent with the NI-DAQ API for Windows 95. The LabWindows/CVI Data Acquisition Library presents a Windows 95 style API for *both* Windows 95 and Windows NT.

If you try to compile existing NI-DAQ programs for Windows NT in LabWindows/CVI, the LabWindows/CVI compiler reports data type mismatches on array and reference parameters in your Data Acquisition function calls. Change the declarations of the variables used as the

reference and array parameters from `int` or `unsigned int` to `short` or `unsigned short`.

Changes to the Function Tree and Function Panel Editors

This chapter discusses the changes to the Function Tree Editor and Function Panel Editor.

The Function Tree and Function Panel Editors are documented in Chapters 3 and 4 of the *LabWindows/CVI Instrument Developers Guide*. The changes documented in this chapter apply to all platforms, unless otherwise marked.

Function Tree Editor

The following changes have been made to the Function Tree Editor.

Maximum Number of Levels Increased to Eight

The maximum number of levels in a function tree has been increased from four to eight.

Create DLL Project (Windows 95/NT Only)

The **Generate DLL Makefile** command in the **Options** menu of the Function Tree Editor window for Windows 3.1 has been replaced by the **Create DLL Project** command for Windows 95 and NT. The **Create DLL Project** command creates a LabWindows/CVI project (`.prj`) file that can be used to create a dynamic link library (`.dll`) from the program file associated with the function panel (`.fp`) file. If there is no program file associated with the `.fp` file, the project is created with a `.c` and `.h` file of the same base name as the `.fp` file.

When you execute this command, you are prompted to enter a pathname for the project file. After the file is written, you are asked if you want to load the project immediately. If you do, your current project is unloaded.

For more information on creating DLLs, see the *Preparing Source Code for Use in a DLL* section in Chapter 1, *Updates to the Programmer Reference Manual* in this document.

Function Panel Editor

The following changes have been made to the Function Panel Editor.

VXI Plug & Play Style

The **VXI Plug & Play Style** command has been added to the **Options** menu in the Function Tree Editor window for Windows 95 and NT. It works in conjunction with the **Create DLL Project** command. When a checkmark appears next to the **VXI Plug & Play Style** command in the menu, the **Create DLL Project** command creates a project that conforms to the rules for VXI*plug&play* instrument driver DLLs. For example, "_32" is appended to the filename of the DLL.

Numeric Control Supports Additional Data Types

You can now use the following data types in a numeric control.

```
int
short
char
unsigned int
unsigned short
unsigned char
double
float
```

Changes to the Programmer's Toolbox

The Programmer's Toolbox is a set of generally useful instrument drivers in the `cvi\toolslib\toolbox` directory. The documentation for the functions is contained in the function panels for each instrument driver. This chapter describes the additions and enhancements to the Programmer's Toolbox at a general level. For detailed information, refer to the function panels.

Additions to the inifile Instrument Driver

You can use the functions of the `inifile` instrument driver to read and write Windows `.ini` style text files. The following additions have been made.

New Functions to Handle DOS/Windows Pathnames

In the `inifile` instrument driver, the original functions for reading and writing string values use backslashes as escape codes for unprintable characters. This causes a problem when you want to read and write DOS/Windows pathnames.

New functions have been added which treat backslashes as any other characters. They are the following.

```
Ini_PutRawString
Ini_GetPointerToRawString
Ini_GetRawStringCopy
Ini_GetRawStringIntoBuffer
```

Easy Tab Instrument Driver Added

You can use functions in the `easytab` instrument driver to display LabWindows/CVI User Interface panels in an overlapped manner with tabs, very similar to a Windows 95 tabbed dialog.

The `easytab` instrument driver is very flexible. You can use it to perform the following actions.

- Create multiple rows of tabs.
- Nest tabbed dialogs within tabbed dialogs.
- Put multiple tabbed dialogs on the same parent panel.

- Show the tabs on the top, bottom, left, or right of the panels.
- Choose whether to stretch or separate the tabs to fit the entire length or width of the panels.
- Set a maximum gap between the tabs.
- Assign underline key accelerators to the tab labels.

The `easytab` instrument driver uses a LabWindows/CVI canvas control to contain the panels and the tabs. When you create a tabbed dialog, you get a control ID that can be used just like any other User Interface Library control ID. The `easytab` instrument driver automatically sizes the panels so that they fit within the canvas control.

You can add panels to the tabbed dialog in either of two ways.

- You can specify a list of handles for panels previously loaded or created with the `LoadPanel` or `NewPanel` function.
- You can specify a list of resource IDs for panels in a .uir file.

New Instrument Driver for Regular Expression Matching

You can use the functions in the `regexpr` instrument driver to search text strings for patterns that match regular expressions.

The driver's high-level function takes a text string and a regular expression and finds a substring which matches the regular expression.

The driver's low-level functions parse a regular expression, match it to the beginning of a text buffer, and then eliminate the parse tree.

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 numbers:

(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.

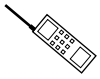
E-Mail Support (currently U.S. only)

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`

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	519 622 9311
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 71 11
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)_____

Version of Excel (look at Excel's About box): 5.0, 5.0c, other_____

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 4.0 Addendum**

Edition Date: **March 1996**

Part Number: **321194A-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, MS 53-02
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
MS 53-02
(512) 794-5678