LabWindows™/CVI™

Programmer Reference Manual

June 2003 Edition
Part Number 323643A-01

Worldwide Technical Support and Product Information

ni.com

National Instruments Corporate Headquarters

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 683 0100

Worldwide Offices

Australia 1800 300 800, Austria 43 0 662 45 79 90 0, Belgium 32 0 2 757 00 20, Brazil 55 11 3262 3599, Canada (Calgary) 403 274 9391, Canada (Montreal) 514 288 5722, Canada (Ottawa) 613 233 5949, Canada (Québec) 514 694 8521, Canada (Toronto) 905 785 0085, Canada (Vancouver) 514 685 7530, China 86 21 6555 7838, Czech Republic 420 2 2423 5774, Denmark 45 45 76 26 00, Finland 385 0 9 725 725 11, France 33 0 1 48 14 24 24, Germany 49 0 89 741 31 30, Greece 30 2 10 42 96 427, India 91 80 51190000, Israel 972 0 3 6393737, Italy 39 02 413091, Japan 81 3 5472 2970, Korea 82 02 3451 3400, Malaysia 603 9131 0918, Mexico 001 800 010 0793, Netherlands 31 0 348 433 466, New Zealand 1800 300 800, Norway 47 0 66 90 76 60, Poland 48 0 22 3390 150, Portugal 351 210 311 210, Russia 7 095 238 7139, Singapore 65 6226 5886, Slovenia 386 3 425 4200, South Africa 27 0 11 805 8197, Spain 34 91 640 0085, Sweden 46 0 8 587 895 00, Switzerland 41 56 200 51 51, Taiwan 886 2 2528 7227, Thailand 662 992 7519, United Kingdom 44 0 1635 523545

For further support information, refer to the *Technical Support and Professional Services* appendix. To comment on the documentation, send email to techpubs@ni.com.

Important Information

Warranty

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

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

National Instruments believes that the information in this document 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

CVI™, DataSocket™, IVI™, National Instruments™, NI™, NI-488.2™, ni.com™, NI-DAQ™, NI-VISA™, and NI-VXI™ are trademarks of National Instruments Corporation.

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

Patents

For patents covering National Instruments products, refer to the appropriate location: **Help»Patents** in your software, the `patents.txt` file on your CD, or `ni.com/patents`.

WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

(1) NATIONAL INSTRUMENTS PRODUCTS ARE NOT DESIGNED WITH COMPONENTS AND TESTING FOR A LEVEL OF RELIABILITY SUITABLE FOR USE IN OR IN CONNECTION WITH SURGICAL IMPLANTS OR AS CRITICAL COMPONENTS IN ANY LIFE SUPPORT SYSTEMS WHOSE FAILURE TO PERFORM CAN REASONABLY BE EXPECTED TO CAUSE SIGNIFICANT INJURY TO A HUMAN.

(2) IN ANY APPLICATION, INCLUDING THE ABOVE, RELIABILITY OF OPERATION OF THE SOFTWARE PRODUCTS CAN BE IMPAIRED BY ADVERSE FACTORS, INCLUDING BUT NOT LIMITED TO FLUCTUATIONS IN ELECTRICAL POWER SUPPLY, COMPUTER HARDWARE MALFUNCTIONS, COMPUTER OPERATING SYSTEM SOFTWARE FITNESS, FITNESS OF COMPILERS AND DEVELOPMENT SOFTWARE USED TO DEVELOP AN APPLICATION, INSTALLATION ERRORS, SOFTWARE AND HARDWARE COMPATIBILITY PROBLEMS, MALFUNCTIONS OR FAILURES OF ELECTRONIC MONITORING OR CONTROL DEVICES, TRANSIENT FAILURES OF ELECTRONIC SYSTEMS (HARDWARE AND/OR SOFTWARE), UNANTICIPATED USES OR MISUSES, OR ERRORS ON THE PART OF THE USER OR APPLICATIONS DESIGNER (ADVERSE FACTORS SUCH AS THESE ARE HEREAFTER COLLECTIVELY TERMED "SYSTEM FAILURES"). ANY APPLICATION WHERE A SYSTEM FAILURE WOULD CREATE A RISK OF HARM TO PROPERTY OR PERSONS (INCLUDING THE RISK OF BODILY INJURY AND DEATH) SHOULD NOT BE RELIANT SOLELY UPON ONE FORM OF ELECTRONIC SYSTEM DUE TO THE RISK OF SYSTEM FAILURE. TO AVOID DAMAGE, INJURY, OR DEATH, THE USER OR APPLICATION DESIGNER MUST TAKE REASONABLY PRUDENT STEPS TO PROTECT AGAINST SYSTEM FAILURES, INCLUDING BUT NOT LIMITED TO BACK-UP OR SHUT DOWN MECHANISMS. BECAUSE EACH END-USER SYSTEM IS CUSTOMIZED AND DIFFERS FROM NATIONAL INSTRUMENTS' TESTING PLATFORMS AND BECAUSE A USER OR APPLICATION DESIGNER MAY USE NATIONAL INSTRUMENTS PRODUCTS IN COMBINATION WITH OTHER PRODUCTS IN A MANNER NOT EVALUATED OR CONTEMPLATED BY NATIONAL INSTRUMENTS, THE USER OR APPLICATION DESIGNER IS ULTIMATELY RESPONSIBLE FOR VERIFYING AND VALIDATING THE SUITABILITY OF NATIONAL INSTRUMENTS PRODUCTS WHENEVER NATIONAL INSTRUMENTS PRODUCTS ARE INCORPORATED IN A SYSTEM OR APPLICATION, INCLUDING, WITHOUT LIMITATION, THE APPROPRIATE DESIGN, PROCESS AND SAFETY LEVEL OF SUCH SYSTEM OR APPLICATION.

Contents

About This Manual

Chapter 1
LabWindows/CVI Compiler

Chapter 2
Using Loadable Compiled Modules

Chapter 3
Compiler/Linker Issues

Chapter 4
Creating and Distributing Release Executables and DLLs

About This Manual

The *LabWindows/CVI Programmer Reference Manual* contains information to help you develop programs in LabWindows™/CVI™. To use this manual effectively, you should be familiar with the *Getting Started with LabWindows/CVI* manual, the *Using LabWindows/CVI* section of the *LabWindows/CVI Help*, Windows, and the C programming language.

Conventions

The following conventions appear in this manual:

» The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box. The **»** symbol also leads you through the Table of Contents in the *LabWindows/CVI Help* to a help topic.

This icon denotes a note, which alerts you to important information.

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

bold Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.

italic Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.

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

`monospace italic` Italic text in this font denotes text that is a placeholder for a word or value that you must supply.

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

Related Documentation

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

- *LabWindows/CVI Help*
- *Getting Started with LabWindows/CVI*
- *LabWindows/CVI Instrument Driver Developers Guide*
- *NI-DAQmx Help*
- *Traditional NI-DAQ Function Reference Help*
- *NI-488.2 Help*
- *NI-VISA Programmer Reference Help*
- *Microsoft Developer Network Online*, Microsoft Corporation, `http://msdn.microsoft.com`
- Harbison, Samuel P. and Guy L. Steele, Jr., *C: A Reference Manual*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1995

1

LabWindows/CVI Compiler

Overview

The LabWindows/CVI compiler is a 32-bit ANSI C compiler. The kernel of the LabWindows/CVI compiler is the lcc ANSI C compiler (© Copyright 1990, 1998 David R. Hanson). The compiler does not generate optimized code but instead focuses on debugging and user protection. Because the compiler is an integral part of the LabWindows/CVI environment and features a limited set of straightforward options, it also is easy to use.

LabWindows/CVI Compiler Specifics

This section describes specific LabWindows/CVI compiler limits, options, defines, and diversions from the ANSI C Standard.

Compiler Limits

Table 1-1 shows the compiler limits for LabWindows/CVI.

Table 1-1. LabWindows/CVI Compiler Limits

Coding Attribute	Limit
Maximum nesting of `#include`	32
Maximum nesting of `#if`, `#ifdef`	16
Maximum number of macro parameters	32
Maximum number of function parameters	256
Maximum number of function parameters with user protection	64
Maximum nesting of compound blocks	32
Maximum size of array/struct types	2^{31}

Compiler Options

To set the LabWindows/CVI compiler options, select **Options»Build Options** in the Workspace window.

Compiler Defines

The LabWindows/CVI compiler accepts compiler defines in the Build Options dialog box, which you can access by selecting **Options»Build Options**.

Compiler defines have the following syntax:

`/Dx` or `/Dx=y`

The variable *x* is a valid C identifier. You can use *x* in source code as a predefined macro. For example, you can use *x* as the argument to the `#if` or `#ifdef` preprocessor directive for conditional compilation. If *y* contains embedded blanks, you must surround it with double quotation marks.

The Predefined Macros dialog box, which you can access by selecting **Predefined Macros** in the Build Options dialog box, contains a list of the macros that LabWindows/CVI predefines. This list includes the name and value of each predefined macro. LabWindows/CVI predefines the macros to help you write platform-dependent code.

Note The default compiler defines string contains the following definition: `/DWIN32_LEAN_AND_MEAN`. This definition reduces the time and memory taken to compile Windows SDK include files.

C Language Non-Conformance

LabWindows/CVI accepts the `#line` preprocessor directive but ignores it.

C Language Extensions

The LabWindows/CVI compiler has several extensions to the C language. The purpose is to make the LabWindows/CVI compiler compatible with external compilers that use common C extensions.

Calling Conventions

You can use the following calling convention qualifiers in function declarations.

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

In Microsoft Visual C/C++ and Borland C/C++, the calling convention normally defaults to __cdecl if you do not use a calling convention qualifier. However, you can set options to cause the calling convention to default to __stdcall. LabWindows/CVI behaves the same way. You can set the default calling convention to either __cdecl or __stdcall in the Build Options dialog box. When you create a new project, the default calling convention is __cdecl.

In the __cdecl calling convention, the calling function is responsible for cleaning up the stack. Functions can have a variable number of arguments in __cdecl.

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, LabWindows/CVI does not honor the qualifier. All compilers pass parameters and return values, except for floating-point and structure return values, in the same way for __stdcall functions.

National Instruments recommends you use the __stdcall calling convention for all functions exported from a DLL, except functions with a variable number of arguments. Visual Basic and other non-C Windows programs expect DLL functions to be __stdcall.

Import and Export Qualifiers

You can 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 both external compilers. The LabWindows/CVI cvidef.h include file defines the following macros, which are designed to work in both external compilers.

```
DLLIMPORT
DLLEXPORT
```

An import qualifier informs the compiler that the symbol is defined in a DLL. Declarations of variables imported from a DLL require import qualifiers, but function declarations do not.

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

C++ 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 does not report an error on multiple definitions of the same typedef identifier, as long as the definitions are identical.

Pragmas

You can use `#pragma` statements to give specific instructions to your compiler.

LabWindows/CVI supports the following types of pragmas:

- User protection
- Other compiler warnings
- Pack pragmas
- Message pragmas

User Protection

```
#pragma EnableLibraryRuntimeChecking
#pragma DisableLibraryRuntimeChecking
#pragma EnableFunctionRuntimeChecking
#pragma DisableFunctionRuntimeChecking
```

For more information on user protection pragmas, refer to the *Disabling Library Protection Errors for Functions* section.

Other Compiler Warnings

Using the following pragmas, you can turn on and off compiler/run-time warnings for detecting uninitialized local variables. The following pragmas operate only on the code they surround.

```
#pragma EnableUninitLocalsChecking
#pragma DisableUninitLocalsChecking
```

Use the following pragmas to turn on and off compiler warnings for detecting assignments in conditional expressions, such as `if`, `while`, `do-while`, and `for`. These pragmas operate on a section of a source code file.

```
#pragma EnableAssignInConditional
#pragma DisableAssignInConditional
```

Pack Pragmas

```
#pragma pack
```

Refer to the *Structure Packing* section of Chapter 3, *Compiler/Linker Issues*, for information about using the `pack` pragma.

You can use the `push` and `pop` modifiers to save and restore structure packing. Saving and restoring structure packing is implemented as a stack, last in–first out, and can be nested arbitrarily.

`push`—Saves the current structure packing. You can use `pop` to specify an identifier to restore this setting. You also can specify a new value for the structure packing rather than use a separate `pack` directive.

The following examples demonstrate how to use the `push` modifier.

```
#pragma pack(push)   /* saves current structure packing */

#pragma pack(push, 1)   /* saves current structure packing and
specifies a new value */

#pragma pack(push, identifier)   /* saves current structure packing
and associates an identifier with it */

#pragma pack(push, identifier, 1)   /* saves current structure
packing, associates an identifier with it, and specifies a new value */
```

`pop`—Restores the last structure packing saved with `push`. You can optionally specify an identifier, in which case the structure packing associated with this identifier is restored. More recently saved structure packings are discarded.

The following examples demonstrate how to use the `pop` modifier.

```
#pragma pack(pop)   /* restores previous structure packing */

#pragma pack(pop, identifier)   /* restores structure packing
associated with identifier */
```

Message Pragmas

```
#pragma message
```

Use message pragmas to output a given message to the Build Errors window. Use either `#pragma message ("some message")` or `#pragma message some message`.

Program Entry Points

You can use `WinMain` instead of `main` as the entry-point function to your program. You might want to use `WinMain` if you plan to link your executable using an external compiler. You must include `windows.h` for the data types that normally appear in the `WinMain` parameter list. The following code 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)
```

Using Low-Level I/O Functions

Many functions in the C compiler libraries are not ANSI C Standard Library functions. In general, LabWindows/CVI implements the ANSI C Standard Library.

The low-level I/O functions `open`, `close`, `read`, `write`, `lseek`, and `eof` are not in the ANSI C Standard Library. You can use these functions along with `sopen` and `fdopen` if you include `lowlvlio.h`.

C Data Types

Table 1-2 shows the data types allowed in LabWindows/CVI.

Table 1-2. Allowable Data Types

Type	Size	Minimum	Maximum
`char`	8	–128	127
`unsigned char`	8	0	255
`short`	16	–32,768	32,767
`unsigned short`	16	0	65,535
`int; long int`	32	-2^{31}	$2^{31} - 1$
`__int64`	64	-2^{63}	$2^{63} - 1$

Table 1-2. Allowable Data Types (Continued)

Type	Size	Minimum	Maximum
`unsigned int`	32	0	$2^{32} - 1$
`unsigned __int64`	64	0	$2^{64} - 1$
`unsigned long`	32	0	$2^{32} - 1$
`float`	32	–3.40282E+38	3.40282E+38
`double; long double`	64	–1.79769E+308	1.79769E+308
`pointers (void *)`	32	N/A	N/A
`enum`	8, 16, or 32	-2^{31}	$2^{31} - 1$

The size of an enumeration type depends on the value of its enumeration constant. In LabWindows/CVI, characters are signed, unless you explicitly declare them unsigned. The types `float` and `double` conform to 4-byte and 8-byte IEEE standard formats.

Converting 16-Bit Source Code to 32-Bit Source Code

If you convert a LabWindows for DOS application to a LabWindows/CVI application, use this section as a guide.

Translation Process

Note Back up any source code you want to continue to use in LabWindows for DOS before you begin the conversion process.

To convert LabWindows for DOS source code so that it runs correctly in LabWindows/CVI, open the source file and select **Options»Translate LW DOS Program**. This command opens the Translate LW DOS Program dialog box, which includes the following options:

- **Select text to translate**—Use this option to specify whether to translate the entire file, translate from the current line, or translate only the selected text.
- **Prompt on every change**—Enable this option to invoke a dialog box each time a change to your source code is necessary. You can make, mark, or skip the change, undo the previous action, or stop the translation in this dialog box.
- **Mark only**—When you enable this option, LabWindows/CVI adds comments to your program but does not make any functional changes to your code.

Once you configure the Translate LW DOS Program dialog box options, select **Start** to begin the automated translation process. You can select **Read Me** to access information about the translation procedure and limitations.

Data Type Size Considerations

If you make a few assumptions about the sizes of data types, little difference exists between a 16-bit compiler and a 32-bit compiler, except for the larger capacity of integers and the larger address space for arrays and pointers.

For example, the code `int x;` declares a 2-byte integer in a 16-bit compiler, such as LabWindows for DOS. In contrast, a 32-bit compiler, such as LabWindows/CVI, handles this code as a declaration of a 4-byte integer. In most cases, this does not cause a problem. The conversion is transparent because functions that use 2-byte integers in LabWindows for DOS use 4-byte integers in LabWindows/CVI. However, this conversion causes a problem when a program performs one of the following actions:

- Passes an array of 16-bit integers to a GPIB, VXI, or data acquisition (DAQ) function

  If you use a 32-bit `int` array to receive a set of 16-bit integers from a device, LabWindows/CVI packs two 16-bit values into each element of the 32-bit array. Any attempt to access the array on an element-by-element basis does not work. Declare the array as `short` instead and make sure any type specifiers that refer to the array have the `[b2]` modifier when you pass them as arguments to Formatting and I/O Library functions.

- Uses an `int` variable in a way that requires the variable to be a 2-byte integer

  For example, if you pass an `int` argument by address to a function in the Formatting and I/O Library, such as a `Scan` source or a `Scan`/`Fmt` target, and the argument matches a `%d[b2]` or `%i[b2]` specifier, the function does not work correctly. Remove the `[b2]` modifier or declare the variable as `short`.

  Conversely, if you pass a `short` argument by address and the argument matches a `%d` or `%i` specifier without the `[b2]` modifier, the function does not work correctly. Add the `[b2]` modifier.

All pointers are 32-bit offsets.

Note The default for `%d` is 2 bytes on a 16-bit compiler and 4 bytes on a 32-bit compiler. In the same way, the default for `int` is 2 bytes on a 16-bit compiler and 4 bytes on a 32-bit compiler. This is why you do not have to make any modifications if the specifier for a variable of type `int` is `%d` without the `[bn]` modifier.

Debugging Levels

You can compile the source modules in your application to include debugging information. If you do so, you can use breakpoints and view or modify variables and expressions while your program is suspended. To set the debugging level, select **Options»Build Options** in the Workspace window.

User Protection

User protection detects invalid program behaviors that LabWindows/CVI cannot otherwise detect during compilation. LabWindows/CVI reports these kinds of invalid program behaviors as user protection errors. When you set the debugging level to Standard or Extended, LabWindows/CVI maintains extra information for arrays, structures, and pointers and uses the information at run time to determine the validity of addresses.

Two groups of user protection errors exist based upon two characteristics: *severity level* and *error category*. In each case, the ANSI C standard states that programs with these errors have undefined behavior. The two severity levels are as follows:

- *Non-fatal* errors include expressions that are likely to cause problems but do not directly affect program execution. The expression is invalid and its behavior is undefined, but execution can continue. Examples include bad pointer arithmetic, attempts to free pointers more than once, and comparisons of pointers to different array objects.
- *Fatal* errors include expressions that LabWindows/CVI cannot execute without causing major problems, such as a general protection fault. For example, dereferencing an invalid pointer value is a fatal error.

Error categories include pointer protection, dynamic memory protection, general protection errors, and library protection.

Array Indexing and Pointer Protection Errors

Pointer protection errors catch invalid operations with pointers and arrays. The errors in this section are grouped by the type of expression that causes the error or the type of invalid pointer involved.

Pointer Arithmetic (Non-Fatal)

Pointer arithmetic expressions involve a pointer subexpression and an integer subexpression. LabWindows/CVI generates an error when the pointer subexpression is invalid or when the

arithmetic operation results in an invalid pointer expression. The following user protection errors involve pointer arithmetic.

- Arithmetic involving an uninitialized pointer
- Arithmetic involving a null pointer
- Out-of-bounds pointer arithmetic (calculation of an array address that results in a pointer value either before the start or past the end of the array)
- Arithmetic involving a pointer to freed memory
- Arithmetic involving an invalid pointer
- Arithmetic involving the address of a non-array object
- Arithmetic involving a pointer to a function
- Array index too large
- Negative array index

Pointer Assignment (Non-Fatal)

LabWindows/CVI generates pointer assignment errors when you assign invalid values to pointer variables. These warnings can help determine when a particular pointer becomes invalid. The following user protection errors involve pointer assignment.

- Assignment of an uninitialized pointer value
- Assignment of an out-of-bounds pointer expression (assignment of an address before the start or past the last element of an array)
- Assignment of a pointer to freed memory
- Assignment of an invalid pointer expression

Pointer Dereferencing (Fatal)

Dereferencing of invalid pointer values is a fatal error because it can cause a memory fault or other serious problems. The following user protection errors involve pointer dereferencing.

- Dereferencing of an uninitialized pointer
- Dereferencing of a null pointer
- Dereferencing of an out-of-bounds pointer (dereferencing using a pointer value before the start or past the end of an array)
- Dereferencing of a pointer to freed memory
- Dereferencing of an invalid pointer expression
- Dereferencing of a data pointer for use as a function
- Dereferencing of a function pointer for use as data
- Dereferencing of a pointer to an *n*-byte type where less than *n* bytes exist in the object

Pointer Comparison (Non-Fatal)

LabWindows/CVI generates pointer comparison errors for erroneous pointer comparison expressions. The following user protection errors involve pointer comparison.

- Comparison involving an uninitialized pointer
- Comparison involving a null pointer
- Comparison involving an invalid pointer
- Comparison of pointers to different objects
- Comparison involving the address of a non-array object
- Comparison of pointers to freed memory

Pointer Subtraction (Non-Fatal)

LabWindows/CVI generates pointer subtraction errors for erroneous pointer subtraction expressions. The following user protection errors involve pointer subtraction.

- Subtraction involving an uninitialized pointer
- Subtraction involving a null pointer
- Subtraction involving an invalid pointer
- Subtraction of pointers to different objects
- Subtraction involving the address of a non-array object
- Subtraction of pointers to freed memory

Pointer Casting (Non-Fatal)

LabWindows/CVI generates a pointer casting error when you cast a pointer expression to type `(AnyType *)`, and not enough space exists for an object of type `AnyType` at the location the pointer expression specifies. This error occurs only when you cast a dynamically allocated object for the first time, such as with the code `(double *) malloc(1)`. In this example, LabWindows/CVI reports the following error: **`Not enough space for casting expression to 'pointer to double'`**.

Dynamic Memory Protection Errors

Dynamic memory protection errors report illegal operations with dynamic memory and corrupted dynamic memory during allocation and deallocation.

Memory Deallocation (Non-Fatal)

LabWindows/CVI generates memory deallocation errors when the pointer is not the result of a memory allocation. The following user protection errors involve memory deallocation.

- Attempt to free an uninitialized pointer
- Attempt to free a pointer to freed memory
- Attempt to free an invalid pointer expression
- Attempt to free a pointer not allocated with `malloc` or `calloc`

Memory Corruption (Fatal)

LabWindows/CVI generates memory corruption errors when a memory allocation/deallocation detects corrupted memory. During each dynamic memory operation, LabWindows/CVI verifies the integrity of the memory blocks it uses in the operation. When you set the debugging level to Extended, LabWindows/CVI thoroughly checks all dynamic memory on each memory operation. LabWindows/CVI generates the following error when it discovers a problem: **`Dynamic memory is corrupt`**.

General Protection Errors

LabWindows/CVI also checks for stack overflow and missing return values. The following errors are general protection errors.

- Stack overflow (fatal)
- Missing return value (non-fatal)

The missing return value error means that a non-void function—a function you do not declare with the `void` return type—returned but did not return a value.

Library Protection Errors

Library functions generate errors when they receive invalid arguments. LabWindows/CVI error checking is sensitive to the requirements of each library function. The following errors involve library protection.

- Passing a null pointer argument to a library function
- Passing an uninitialized pointer argument to a library function
- Passing a pointer to freed memory to a library function
- Passing an argument that is too small
- Passing by reference a scalar argument to a library function that expects an array
- Passing a string argument that does not have a terminating null character
- Passing a string to a library function that expects a character reference parameter

LabWindows/CVI library functions return error codes in a variety of cases. If you enable the **Run»Break on»Library Errors** option in the Workspace window, LabWindows/CVI suspends execution after a library function returns one of these errors. A message appears that displays the name of the function and either the return value or a string that explains why the function failed.

Disabling User Protection

Occasionally, you might want to disable user protection to avoid run-time errors that do not cause problems in your program.

Disabling Protection Errors at Run Time

You can use the `SetBreakOnProtectionErrors` function in the Utility Library to programmatically control whether LabWindows/CVI suspends execution when it encounters a protection error. This function does not affect the **Run»Break on»Library Errors** option.

Disabling Library Errors at Run Time

The **Run»Break on»Library Errors** option in the Workspace window lets you choose whether LabWindows/CVI suspends execution when a library function returns an error code. The option takes effect when you start executing the project. You can override the initial setting in your program by using the `SetBreakOnLibraryErrors` function in the Utility Library. Using this function does not affect the reporting of other types of library protection errors.

Disabling Protection for Individual Pointer

You can disable pointer checking for a particular pointer by casting it first to an arithmetic type and then back to its original type, as shown in the following macro:

```
#define DISABLE_RUNTIME_CHECKING(ptr)((ptr) = (void *)
             ((unsigned)(ptr)))
{
    char *charPointer;
    /* run-time checking is performed for charPointer before
    this line */
    DISABLE_RUNTIME_CHECKING(charPointer);
    /* no run-time checking is performed for charPointer after
    this line */
}
```

This macro can be useful if LabWindows/CVI reports erroneous run-time errors because you set a pointer to dynamic memory in a source module and then you resize this pointer in an object module. The following steps describe how this error occurs.

1. You declare a pointer in a source module that you compile with debugging enabled. You then assign to the pointer an address that `malloc` or `calloc` returns.

   ```
   AnyType *ptr;
   ptr = malloc(N);
   ```

2. You reallocate the pointer in an object module so that it points to the same location in memory as before. This might occur if you call the `realloc` function or free the pointer and then reassign it to memory that you allocate with `malloc`.

   ```
   ptr = realloc(ptr, M); /* M > N */
   ```

   or

   ```
   free(ptr);
   ptr = malloc(M);
   ```

3. You use the same pointer in a source module you compile with debugging enabled. At this point, LabWindows/CVI still expects the pointer to point to a block of memory of the original size (`N`).

   ```
   *(ptr+(M-1))/* This generates a fatal run-time error, */
   /* even though it is a legal expression. */
   ```

To prevent this error, use the `DISABLE_RUNTIME_CHECKING` macro to disable checking for the pointer after you allocate memory for it in the source module.

```
ptr = malloc(N);
DISABLE_RUNTIME_CHECKING(ptr);
```

Disabling Library Protection Errors for Functions

You can disable or enable library protection errors by placing pragmas in the source code. LabWindows/CVI ignores these pragmas when you compile without debugging information, that is, if you set the active configuration to the release configuration. For example, the following two pragmas enable and disable library checking for all the function declarations that occur after the pragma within a header or source file. The pragmas affect only the functions declared in the file in which the pragmas occur. These pragmas do not affect nested include files.

```
#pragma EnableLibraryRuntimeChecking
#pragma DisableLibraryRuntimeChecking
```

The following pragmas enable and disable library checking for a particular function. You must declare the function before the pragma occurs.

```
#pragma EnableFunctionRuntimeChecking function
#pragma DisableFunctionRuntimeChecking function
```

These two preceding pragmas enable and disable run-time checking for a particular library function throughout the module in which they appear. You can use these pragmas to override the effects of the `EnableLibraryRuntimeChecking` and `DisableLibraryRuntimeChecking` pragmas for individual functions. If both of these pragmas occur in a module for the same function, LabWindows/CVI uses only the last occurrence.

Note These pragmas affect all protection, including run-time checking of function arguments, for all calls to a specific library function. To disable breaking on errors for a particular call to a library function, use the Utility Library `SetBreakOnLibraryErrors` function. To disable the run-time checking of argument expressions for a particular call to a library function, use the Utility Library `SetBreakOnProtectionErrors` function.

Note You cannot use pragmas to disable protection for functions in the statically linked libraries including the User Interface, RS-232, TCP Support, DDE Support, Formatting and I/O, Utility, and ANSI C Libraries unless you place the `DisableLibraryRuntimeChecking` pragma at the top of the library header file.

Details of User Protection

Pointer Casting

A cast expression consists of a left parenthesis, a type name, a right parenthesis, and an operand expression. The cast causes the compiler to convert the operand value to the type that appears within the parentheses.

C programmers occasionally need to cast a pointer to one data type to a pointer to another data type. Because LabWindows/CVI does not restructure the user protection information for each cast expression, certain types of cast expressions implicitly disable run-time checking for the pointer value. In particular, casting a pointer expression to the following types disables run-time checking on the resulting value.

Pointer to a pointer:	`(AnyType **) PointerExpression`
Pointer to a structure:	`(struct AnyStruct *) PointerExpression`
Pointer to an array:	`(AnyType (*)[]) PointerExpression`
Any non-pointer type:	`(unsigned) PointerExpression`, `(int) PointerExpression`, and so on

Note An exception exists. Casts that you apply implicitly or explicitly to the `void *` values you obtain from `malloc` or `calloc` do not disable user protection.

Casting a pointer to one arithmetic type to a pointer to a different arithmetic type, such as `(int *)`, `(unsigned *)`, `(short *)`, and so on, does not affect run-time checking on the resulting pointer nor does casting a pointer to a void pointer `(void *)`.

Dynamic Memory

LabWindows/CVI provides run-time error checking for pointers and arrays in dynamically allocated memory.

You can use the ANSI C Library functions `malloc` or `calloc` to allocate dynamic memory. These functions return `void *` values that you must cast to some other type before you can use the memory. During program execution, LabWindows/CVI uses the first such cast on the return value of each call to these functions to determine the type of the object that will be stored in the dynamic memory. Subsequent casts to different types can disable checking on the dynamic data, as explained in the *Pointer Casting (Non-Fatal)* section.

You can use the `realloc` function to resize dynamically allocated memory. This function increases or decreases the size of the object associated with the dynamic memory. LabWindows/CVI adjusts the user protection information accordingly.

Avoid Unassigned Dynamic Allocation in Function Parameters

The LabWindows/CVI run-time error checking mechanism dynamically allocates data to keep track of pointers that you dynamically allocate in your program. When you no longer use the pointers, LabWindows/CVI uses garbage collection to deallocate the pointers' corresponding dynamic memory.

A case exists where the garbage collection fails to retrieve all the memory that the run-time error checking mechanism allocated. This case occurs when the all of the following points are true:

- You pass the return value of one function to another function.
- The return value is a pointer to dynamically allocated memory.
- You do not assign the pointer to a variable in the argument expression.

The following code is an example of such a case.

```
MyFunc (1, 2, malloc(7));
```

This call passes the return value from `malloc` to `MyFunc` but does not assign the return value to a variable. If you make this call repeatedly in your program with run-time checking enabled, you lose a small amount of memory each time.

Change the code as follows to avoid this problem.

```
void *p;
MyFunc (1, 2, p = malloc(7));
```

The following code also works and uses better programming style.

```
void *p;
p = malloc(7);
MyFunc (1, 2, p);
```

Library Functions

The LabWindows/CVI library functions that take pointer arguments or that return pointers incorporate run-time checking for those arguments and return values. However, you must be careful when passing arguments to library functions that have `void *` parameters, such as `GetCtrlAttribute` and `GetCtrlVal` in the User Interface Library and `memcpy` and `memset` in the ANSI C Library. If you use a `void *` cast when you pass an argument to a function that expects a variably typed argument, you disable run-time checking for that argument, as shown in the following examples:

```
{
    int value;
    GetCtrlVal(panel, ctrl, &value); /* CORRECT */
    GetCtrlVal(panel, ctrl, (void *)&value); /* INCORRECT */
}

{
    char *names[N], *namesCopy[N];
    memcpy(namesCopy, names, sizeof (names)); /* CORRECT */
    memcpy((void *)namesCopy, (void *)names, sizeof names);
    /* INCORRECT */
}
```

Unions

LabWindows/CVI performs only minimal checks for union type variables. If a union contains pointers, arrays, or structs, LabWindows/CVI does not maintain user protection information for those objects.

Stack Size

Your program uses the stack to pass function parameters and store automatic local variables. You can set the maximum stack size by selecting **Options»Build Options** in the Workspace window. LabWindows/CVI supports the following stack size range:

- Minimum = 100 KB
- Default = 250 KB
- Maximum = 1 MB

If you enable the **Detect uninitialized local variables at runtime** option in the Build Options dialog box, LabWindows/CVI might use extra stack space. You can adjust your maximum stack size to avoid a stack overflow.

Include Paths

The **Include Paths** option in the Environment dialog box invokes a dialog box in which you can list paths that the compiler uses when searching for header files with simple pathnames. The Include Paths dialog box has two lists of paths in which to search for include files, one for include paths specific to the project and one for paths not specific to the project. LabWindows/CVI saves the list of paths specific to the project with the project file. LabWindows/CVI saves the list of paths not specific to the project from one session to another on the same machine, regardless of the project. When you install VXI*plug&play* instrument drivers, the installation program places the include files for the drivers in a specific VXI*plug&play* include directory. If you install IVI instrument drivers, the installation program places the include files for those drivers in a specific IVI include directory. LabWindows/CVI also searches those directories for include files.

To access the Environment dialog box, select **Options»Environment**.

Include Path Search Precedence

LabWindows/CVI searches for include files in the following locations and in the following order:

1. Project list
2. Project-specific include paths
3. Non-project-specific include paths
4. The paths listed in the Instrument Directories dialog box
5. The subdirectories under the `CVI70\toolslib` directory
6. The `CVI70\instr` directory
7. The `CVI70\include` directory
8. The `CVI70\include\ansi` directory
9. The IVI include directory
10. The VXI*plug&play* include directory
11. The `CVI70\sdk\include` directory

2

Using Loadable Compiled Modules

About Loadable Compiled Modules

Several methods exist for using compiled modules in LabWindows/CVI. You can load compiled modules directly into the LabWindows/CVI environment as instrument driver programs or as user libraries, so they are accessible to any project. You can list compiled modules in your project so they are accessible only within that project. You can use compiled modules dynamically in your program with `LoadExternalModule`, `RunExternalModule`, and `UnloadExternalModule`. Any compiled module you use in LabWindows/CVI must be in one of the following forms:

- A `.obj` file that contains one object module
- A `.lib` file that contains one or more object modules
- A `.dll` file that contains a Windows DLL

You can create any of these compiled modules in LabWindows/CVI or in a compatible external compiler.

Advantages and Disadvantages of Using Loadable Compiled Modules in LabWindows/CVI

Using compiled modules in LabWindows/CVI has the following advantages:

- Compiled modules you generate in external compilers can run faster than source modules because of optimization. Compiled modules do not contain the debugging and user protection code that LabWindows/CVI generates when it compiles source modules.
- If an instrument driver program file is a source module and the source module is not in the project, LabWindows/CVI recompiles it each time you load the instrument driver. LabWindows/CVI does not recompile compiled modules each time you load an instrument driver.
- You can dynamically load compiled modules, but not source modules, in stand-alone executables.

- You can install compiled modules, but not source modules, into the Library Tree.
- You can provide libraries for other developers without giving them access to your source code.

Using compiled modules in LabWindows/CVI has the following disadvantages:

- You cannot debug compiled modules. Because compiled modules do not contain any debugging information, you cannot set breakpoints or view variable values.
- Compiled modules do not include run-time error checking or user protection.

Using a Loadable Compiled Module as an Instrument Driver Program File

An instrument driver is a set of high-level functions with graphical function panels to make programming easier. The instrument driver encapsulates many low-level operations, such as data formatting and GPIB, RS-232, and VXI communication, into intuitive, high-level functions. An instrument driver usually controls a physical instrument, but the instrument driver also can be a software utility.

To develop and debug an instrument driver, load its program file into LabWindows/CVI as a source file. After you finish debugging, you can compile the program file into an object file or a Windows DLL. The next time you load the instrument driver, LabWindows/CVI loads the compiled module, which loads and runs faster than the source module.

If the instrument driver program file is a compiled module, it must adhere to the requirements outlined in Chapter 3, *Compiler/Linker Issues*.

Using a Loadable Compiled Module as a User Library

You can install your own libraries into the Library Tree. A user library has the same form as an instrument driver. You can load as a user library anything that you can load into the **Instrument** menu, provided the program is in compiled form. The main difference between modules you load as instrument drivers and those you load as user libraries is that you can unload instrument drivers using the **Instrument»Unload** command, but you cannot unload user libraries. You cannot edit and recompile user libraries while they are loaded.

To install a user library, right-click the Library Tree and select **Customize Library Menu**. The next time you run LabWindows/CVI, the libraries load automatically and appear in the Library Tree.

You can develop a user library module to provide support functions for instrument drivers or any other modules in your project. By installing a module through the **Customize Library Menu** command, you ensure that the library is always available in the LabWindows/CVI development environment. If you do not want to develop function panels for the library, create a `.fp` file without any classes or functions. In that case, LabWindows/CVI loads the library at startup but does not include the library name in the Library Tree.

User libraries must adhere to the requirements outlined in Chapter 3, *Compiler/Linker Issues*.

Using a Loadable Compiled Module in the Project List

You can include compiled modules directly in the project list.

Note To use a DLL in your project, you must include the DLL import library `(.lib)` file rather than the DLL in the project list.

Compiled modules must adhere to the requirements outlined in Chapter 3, *Compiler/Linker Issues*.

Using a Loadable Compiled Module as an External Module

You can load a compiled module dynamically from your program. A module you load dynamically is called an external module. You can load, execute, and unload this external module programmatically using `LoadExternalModule`, `GetExternalModuleAddr`, and `UnloadExternalModule`.

While you develop and debug the external module, you can list it in the project as a source file. After you finish debugging the module, you can compile it into an object file or a Windows DLL. External modules must adhere to the requirements outlined in Chapter 3, *Compiler/Linker Issues*.

3

Compiler/Linker Issues

This chapter describes the different kinds of compiled modules available under LabWindows/CVI and includes programming guidelines for modules you generate with external compilers.

The LabWindows/CVI compiler is compatible with two external 32-bit compilers: Microsoft Visual C/C++ and Borland C/C++. This manual refers to these two compilers as the compatible external compilers.

In LabWindows/CVI, you can perform the following actions:

- Load 32-bit DLLs using the standard import library mechanism
- Create 32-bit DLLs and DLL import libraries
- Create library files and object files
- Call the LabWindows/CVI libraries from executables or DLLs created with either of the compatible external compilers
- Create object files, library files, and DLL import libraries that the compatible external compilers can use
- Load object files, library files, and DLL import libraries created with either of the compatible external compilers
- Call Windows Software Development Kit (SDK) functions

Loading 32-Bit DLLs

LabWindows/CVI can load 32-bit DLLs. LabWindows/CVI links to DLLs through the standard 32-bit DLL import libraries that you generate when you create 32-bit DLLs with any of the compilers. Because LabWindows/CVI links to DLLs in this way, you cannot specify a DLL file directly in your project. You must specify the DLL import library file instead.

DLLs for Instrument Drivers and User Libraries

LabWindows/CVI does not directly associate DLLs with instrument drivers or user libraries. However, LabWindows/CVI can associate instrument drivers and user libraries with DLL import libraries. Each DLL must have a DLL import library (`.lib`) file. In general, if the program for an instrument driver or user library is in the form of a DLL, you must place the DLL import library in the same directory as the function panel (`.fp`) file. The DLL import

library specifies the name of the DLL that LabWindows/CVI searches for using the standard Windows DLL search algorithm.

LabWindows/CVI makes an exception to facilitate using VXI*plug&play* instrument driver DLLs. When you install a VXI*plug&play* instrument driver, the installation program does not place the DLL import library in the same directory as the `.fp` file. If a `.fp` file is in the VXI*plug&play* directory, LabWindows/CVI searches for an import library in the VXI*plug&play* library directory before it looks for a program file in the directory of the `.fp` file, unless you list the program file in the project.

LabWindows/CVI makes the same exception for IVI instrument driver DLLs. If the `.fp` file is in the `IVI\Drivers` directory, LabWindows/CVI searches for an import library in the `IVI\Drivers\Lib` directory before it looks for a program file in the directory of the `.fp` file, unless you list the program file in the project.

Using the LoadExternalModule Function

When you use 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.

DLL Path (.pth) Files Not Supported

The DLL import library contains the filename 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 2000/NT/XP/Me/98.

16-Bit DLLs Not Supported

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

DllMain

Each DLL can have a `DllMain` function. The Borland compiler uses `DllEntryPoint` as the name for this function. The operating system calls the `DllMain` function with various messages. To generate the template for a `DllMain` function, select **Edit»Insert Construct** in a Source window.

Use caution when inserting code in the `PROCESS_ATTACH` and `PROCESS_DETACH` cases. In particular, avoid calling into other DLLs in these two cases. The order in which Windows initializes DLLs at startup and unloads them at process termination is not well defined. Thus, the DLLs you want to call might not be in memory when your `DllMain` receives the `PROCESS_ATTACH` or `PROCESS_DETACH` message.

It is always safe to call into the LabWindows/CVI Run-time Engine in a `DllMain` function as long as you do so before calling `CloseCVIRTE`.

Releasing Resources when a DLL Unloads

When a program terminates, the operating system disposes resources that your DLL allocates. If your DLL remains loaded throughout program execution, the DLL does not need to dispose resources explicitly when the system unloads the DLL at program termination. However, if the program unloads your DLL during program execution, it is a good idea for your DLL to dispose of any resources it allocates. The DLL can release resources in the `DllMain` function in response to the `PROCESS_DETACH` message. Additionally, the DLL can release resources in a function that it registers with the ANSI C `atexit` function. The system calls the function you register when the DLL receives the `PROCESS_DETACH` message.

If your DLL calls into the LabWindows/CVI Run-time Engine DLL, it can allocate resources such as user interface panels. If a program unloads your DLL during execution, you might want to dispose these resources by calling functions such as `DisposePanel` in the LabWindows/CVI Run-time Engine. On the other hand, as explained in the preceding section, it is generally unsafe to call into other DLLs in response to the `PROCESS_DETACH` message.

To solve this dilemma, you can use the `CVIRTEHasBeenDetached` function in the Utility Library. It is always safe to call the `CVIRTEHasBeenDetached` function. `CVIRTEHasBeenDetached` returns FALSE until the main Run-time Engine DLL, `cvirte.dll`, receives the `PROCESS_DETACH` message. Consequently, if `CVIRTEHasBeenDetached` returns FALSE, your DLL can safely call functions in the LabWindows/CVI Run-time Engine to release resources.

Note `cvirte.dll` contains the User Interface, Utility, Formatting and I/O, RS-232, ANSI C, TCP Support, and DDE Support Libraries.

Generating an Import Library

If you do not have a DLL import library, you can generate an import library in LabWindows/CVI. You must have an include file that contains the declarations of all the functions and global variables you want to access from the DLL. The calling conventions of the function declarations in the include file must match the calling conventions of the functions in the DLL. For example, if the DLL exports functions using the `__stdcall` calling convention, the function declarations in the include file must contain the `__stdcall` keyword. Load the include file into a Source window and select **Options»Generate DLL Import Library**.

Compatibility with External Compilers

LabWindows/CVI can be compatible at the object code level with both of the compatible external compilers, Microsoft Visual C/C++ and Borland C/C++. Because these compilers are not compatible with each other at the object code level, LabWindows/CVI can be compatible with only one external compiler at a time. This manual refers to the compiler with which your copy of LabWindows/CVI is currently compatible as the current compatible compiler.

Choosing the Compatible Compiler

You can see which compatible compiler is active in LabWindows/CVI by selecting **Options» Build Options**. To change the compatible compiler, select **Visual C/C++** or **Borland C/C++** from **Compatibility with** in the Build Options dialog box.

Object Files, Library Files, and DLL Import Libraries

If you create an object file, library file, or DLL import library in LabWindows/CVI, you can use the file only in the current compatible compiler.

If you load an object file, library file, or DLL import library file in LabWindows/CVI, you must have created the file in the current compatible compiler. If you have a DLL but you do not have a compatible DLL import library, LabWindows/CVI reports an error when you attempt to link your project.

To create a compatible import library, you must have an include file that contains the declarations of all the functions and global variables you want to access from the DLL. Load the include file into a Source window and select **Options»Generate DLL Import Library**.

Make sure the calling conventions of the function declarations in the include file match the calling conventions of the functions in the DLL. Whereas DLLs usually export functions with the `__stdcall` calling convention, the `__stdcall` keyword is sometimes missing from the function declarations in the associated include files. If you generate an import library from an include file that does not agree with the calling convention the DLL uses, you can successfully build a project that contains the import library, but LabWindows/CVI usually reports a general protection fault when you run the project.

Compatibility Issues in DLLs

In general, you can use a DLL without regard to the compiler you used to create it. Only the DLL import library must be created for the current compatible compiler. Some cases exist, however, in which you cannot call a DLL that you created using one compiler from an executable or DLL that you created using another compiler. If you want to create DLLs that you can use in different compilers, design the Application Programming Interface (API) for your DLL to avoid such problems. The DLLs that external compilers create are not fully compatible in the areas discussed in this section.

Structure Packing

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

If your DLL API uses structures, you can guarantee compatibility among the different compilers by using the `pack` pragma to specify a specific maximum alignment factor. Place this pragma in the DLL include file, before the definitions of the structures. You can choose any alignment factor. After the structure definitions, reset the maximum alignment factor back to the default, as in the following example:

```
#pragma pack (4) /* set maximum alignment to 4 */
typedef struct {
    char a;
    int b;
} MyStruct1;
typedef struct {
    char a;
    double b;
} MyStruct2;
#pragma pack () /* reset max alignment to default */
```

LabWindows/CVI predefines the `__DEFALIGN` macro to the default structure alignment of the current compatible compiler.

Bit Fields

Borland C/C++ uses the smallest number of bytes necessary to hold the bit fields you specify in a structure. Microsoft Visual C/C++ always uses 4-byte elements. You can force compatibility by adding a dummy bit field of the correct size to pad the set of contiguous bit fields so that they fit exactly into a 4-byte element, as in the following 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`. To solve this problem, change your DLL API so that it uses output parameters instead of return values for `double` and `float` scalars.

Returning Structures

For functions you do not declare with the `__stdcall` calling convention, the compilers return structures using different mechanisms. For functions you declare with `__stdcall`, the compilers return structures in the same way, except for 8-byte structures. National Instruments recommends that you design your DLL API to use structure output parameters instead of structure return values.

Enum Sizes

The compilers always use 4 bytes to represent the largest `enum` value.

Long Doubles

In Borland C/C++ and Borland C++ Builder, `long double` values are 10 bytes. In Microsoft Visual C/C++, they are 8 bytes. In LabWindows/CVI, they are always 8 bytes. Avoid using `long double` in your DLL API.

Differences between LabWindows/CVI and External Compilers

LabWindows/CVI does not work with all the non-ANSI extensions each external compiler provides. 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 differences are the most important ones you might encounter.

- `wchar_t` is 2 bytes 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.
- LabWindows/CVI does not define `_MSC_VER` or `__BORLANDC__`. The external compilers each define one of these macros. If you import code originally developed under one of these external compilers to LabWindows/CVI, you might have to manually define one of these macros.

External Compiler Versions Supported

The following versions of each external compiler work with LabWindows/CVI.

- Microsoft Visual C/C++, version 2.2 or later
- Borland C/C++, version 4.51 or later
- C++ Builder, version 4.0 or later

Required Preprocessor Definitions

When you use an external compiler to compile source code that includes any of the LabWindows/CVI include files, add the following definition to your preprocessor definitions:

```
_NI_mswin32_
```

Multithreading and LabWindows/CVI Libraries

All the LabWindows/CVI libraries are multithreaded safe when used inside or outside of the LabWindows/CVI development environment.

For information about using the LabWindows/CVI User Interface Library in a multithreaded program, refer to *Different Approaches to Multithreaded User Interface Programming* in the *LabWindows/CVI Help*.

Using LabWindows/CVI Libraries in External Compilers

You can use the LabWindows/CVI libraries in either of the compatible external compilers. You can create executables and DLLs that call the LabWindows/CVI libraries. LabWindows/CVI ships with the run-time DLLs that contain all the libraries. Executable files you create in LabWindows/CVI also use these DLLs. The `CVI70\extlib` directory contains DLL import libraries and a startup library, all of which are compatible with your external compiler. Never use the `.lib` files in the `CVI70\bin` directory in an external compiler.

You always must 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 Support Library */
/* TCP Support Library */
/* Utility Library */
```

You can add the following static library file from `CVI70\extlib` to your external compiler project.

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

You can add the following DLL import library files from `CVI70\extlib` to your external compiler project.

```
gpib.lib/* GPIB/GPIB 488.2 Library*/
dataacq.lib/* Traditional NI-DAQ Library*/
visa.lib/* VISA Library*/
nivxi.lib/* VXI Library*/
ivi.lib/* IVI Library*/
nidaqmx.lib/* NI-DAQmx Library*/
cviauto.lib/* ActiveX Library*/
```

If you use an instrument driver that makes references to both the GPIB/GPIB 488.2 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*/
```

If you use an external compiler that requires a `WinMain` entry point, the following optional library allows you to define only `main` in your program.

```
cviwmain.lib /* contains a WinMain() function that*/
/* calls main() */
```

Include Files for ANSI C Library and LabWindows/CVI Libraries

The `cvirt.lib` import library contains symbols for all 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 `CVI70\include` directory, the LabWindows/CVI ANSI C include files are in the `CVI70\include\ansi` directory. Thus, you can specify `CVI70\include` as an include path in your external compiler while at the same time using the external compiler's version of the ANSI C include files.

Note Use the external compiler's ANSI C include files only when you compile a source file that you intend to link using the external compiler. If you intend to link the file in LabWindows/CVI, use the LabWindows/CVI ANSI C include files. This is true regardless of which compiler you use to compile the source file.

For more information about using ANSI C Standard Library include files and LabWindows/CVI library include files, refer to the *Setting Up Include Paths for LabWindows/CVI, ANSI C, and SDK Libraries* section.

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.

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.

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 you load from the user interface resource (`.uir`) file.

When you link your program in an external compiler, the external compiler does not make such a table available to the User Interface Library. Complete the following steps to use LabWindows/CVI to generate a source file that contains the necessary table.

1. Create a LabWindows/CVI project that contains the `.uir` files your program uses, if you do not already have one.
2. Select **Build»External Compiler Support** in the Workspace window to open the External Compiler Support dialog box.
3. Set the **UIR Callbacks** control to **Source File** and enter the pathname of the source file you want to generate. When you click the **Create** button, LabWindows/CVI generates the source file with a table that contains the names of all the callback functions referenced in all the `.uir` files in the project. When you modify and save any of these `.uir` files, LabWindows/CVI regenerates the source file to reflect the changes.
4. Include this source file in the external compiler project you use to create the executable.
5. You must call `InitCVIRTE` at the beginning of your `main` or `WinMain` function.

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 might want to link to non-static callback functions that the DLL contains but does not export. You can do this by calling `LoadPanelEx` and `LoadMenuBarEx`. When you pass the DLL module handle to `LoadPanelEx` and `LoadMenuBarEx`, the User Interface Library searches the table of callback functions the DLL contains before searching the table that the executable contains.

If you create your DLL in LabWindows/CVI, LabWindows/CVI includes the table of functions in the DLL automatically. If you create your DLL using an external compiler, you must generate a source file that contains the necessary table as described in this section.

Resolving References from Modules Loaded at Run Time

Note This section does not apply to you unless you use `LoadExternalModule` or `LoadExternalModuleEx` to load object or static library files.

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 LabWindows/CVI Run-time Engine, from the executable, or from previously loaded run-time modules. A table of these symbols must be available in the executable. When you link your program in LabWindows/CVI, LabWindows/CVI automatically includes a symbol table.

When you link your program in an external compiler, the external compiler does not make such a table available to the Utility Library. Refer to the *Resolving References to LabWindows/CVI Run-Time Engine* section and the *Resolving References to Symbols Not in Run-Time Engine* section for ways to create the symbol table.

Resolving References to LabWindows/CVI Run-Time Engine

LabWindows/CVI makes available two source files that contain symbol table information for the LabWindows/CVI libraries that are in Run-time Engine DLLs.

- Include `CVI70\extlib\refsym.c` in your external compiler project if your run-time modules refer to any symbols in the User Interface, Formatting and I/O, RS-232, DDE Support, TCP Support, or Utility Library.
- Include `CVI70\extlib\arefsym.c` in your external compiler project if your run-time modules refer to any symbols in the ANSI C Library.

Resolving References to Symbols Not in Run-Time Engine

If your run-time modules refer to any other symbols from your executable, you must use LabWindows/CVI to generate an object file that contains a table of those symbols.

1. Create an include file that contains complete declarations of all the symbols your run-time modules reference from the executable. The include file can contain nested `#include` statements and can contain executable symbols that your run-time modules do not refer to. If your run-time module references any of the commonly used Windows SDK functions, you can use the `CVI70\sdk\include\basicsdk.h` file.
2. Select **Build»External Compiler Support** in the Workspace window to open the External Compiler Support dialog box.
3. Enable the **Using LoadExternalModule to Load Object and Static Library Files** option.
4. Enable the **Other Symbols** option if it is not already enabled.
5. Enter the pathname of the include file in the **Header File** control.

6. Enter the pathname of the object file to generate in the **Object File** control.
7. Click the **Create** button to the right of **Object File**.
8. 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.

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

When you load an object or static library file from a DLL, you might want to resolve references from that module using global symbols the DLL contains but does not export. You can do this by calling `LoadExternalModuleEx`. When you pass the DLL module handle to `LoadExternalModuleEx`, the Utility Library searches the symbol table the DLL contains before searching the table that the executable contains.

If you create your DLL in LabWindows/CVI, LabWindows/CVI includes the table of symbols in the DLL automatically. If you create your DLL using an external compiler, the external compiler does not make such a table available to the Utility Library. Thus, when you use an external compiler, you must include in your DLL one or more object files that contain the necessary symbol tables. You must call `InitCVIRTE` and `CloseCVIRTE` in your `DllMain` function.

Calling InitCVIRTE and CloseCVIRTE

If you link an executable or DLL in an external compiler, you must call the `InitCVIRTE` function at the beginning of your `main`, `WinMain`, or `DllMain` function.

For an executable using `main` as the entry point, your code must include the following segment:

```
#include <cvirte.h>
int main (int argc, char *argv[])
{
    if (InitCVIRTE(0, argv, 0) == 0)
        return (-1); /* out of memory */
    /* your other code */
}
```

For an executable using `WinMain` as the entry point, your code must include the following segment:

```
#include <cvirte.h>
int __stdcall WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance
              LPSTR lpszCmdLine, int nCmdShow)
{
    if (InitCVIRTE(hInstance, 0, 0) == 0)
```

```
        return (-1); /* out of memory */
    /* your other code */
}
```

For a DLL, you also have to call `CloseCVIRTE` in `DllMain`. The code must include the following segment:

```
#include <cvirte.h>
int __stdcall DllMain (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
              lpvReserved)
{
    switch (fdwReason)
        {
        case DLL_PROCESS_ATTACH:
           if (InitCVIRTE (hinstDLL, 0, 0) == 0)
              return 0; /* out of memory */
           /* your other ATTACH code */
           break;
        case DLL_PROCESS_DETACH:
           /* your other DETACH code */
           CloseCVIRTE ();
           break;
        }
    return 1;
}
```

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

Using Object and Library Files in External Compilers

When you use an external compiler to link a project that contains object or static library files created in LabWindows/CVI, keep the following points in mind.

Default Library Directives

Most compilers insert default library directives in the object and library files they generate. A default library directive tells the linker to automatically include a named library in the link. Normally, the directive refers to the name of C library files. If no files in the link contain a default library directive and the linker does not explicitly include a C library in the link, the linker reports unresolved function references in the object modules.

Object and static library files that LabWindows/CVI creates do not contain a default library directive. This has different implications for Microsoft Visual C/C++'s compiler and Borland C/C++ and C++ Builder.

Microsoft Visual C/C++—Default Library Directives

If you include in your project at least one object file that contains a default library directive, the Visual C linker uses that library to resolve references in all object and library files, even the files you create in LabWindows/CVI. Object files you create in Visual C usually contain default library directives.

If you do not include in your project any object files or libraries you create in Visual C, you can add the following Visual C libraries to the project to avoid link errors.

```
libc.lib
oldnames.lib
```

In the Visual C development environment, add these library names by selecting **Input** in the **Link** tab of the Project Settings dialog box.

Borland C/C++ and C++ Builder—Default Library Directives

No problems exist with the absence of default library directives when you use the Borland compilers.

Borland Static versus Dynamic C Libraries

When you link a Borland C/C++ or C++ Builder project that contains object or static library files you create in LabWindows/CVI, it is a good idea to configure the Borland project to use the static version of the Borland C libraries.

If you choose to use the dynamic C libraries, you must compile the LabWindows/CVI object modules with the `_RTLDLL` macro. You must define the `_RTLDLL` macro in your source code before including any of the Borland C header files.

Borland C/C++ Incremental Linker

You cannot use your LabWindows/CVI object or static library files in the Borland C compiler if you choose to use the incremental linker. Turn off the **Use Incremental Linker** option in the Borland C/C++ compiler.

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

When you use a compatible external compiler to create an object or library file for use in LabWindows/CVI, you must use the include files in the `CVI70\include` and `CVI70\include\ansi` directories. Ensure that these directories have priority over the default paths for the compiler's C library include files.

You must choose the compiler options carefully. LabWindows/CVI is compatible with the default options for each compiler.

Microsoft Visual C/C++ Defaults

LabWindows/CVI is compatible with all the defaults.

You must *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++ and C++ Builder Defaults

LabWindows/CVI is compatible with all the defaults.

You must *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)
`-fp` (Correct Pentium FDIV Flaw)

Creating Executables in LabWindows/CVI

You can create true 32-bit Windows executables in LabWindows/CVI. The LabWindows/CVI run-time libraries come in DLL form. Stand-alone executables you create in LabWindows/CVI and executables you create in external compilers use the same DLLs. If you run more than one program at a time, Windows loads only one copy of the DLL.

To create a stand-alone executable suitable for distribution, you first must select **Build»Target Type»Executable** in the Workspace window. Next, you must select **Build»Configuration»Release**. After you have done this, the **Create Release Executable** command appears in the **Build** menu. Use the **Create Release Executable** command to create an executable suitable for distribution. You can set the name of the executable as well as other options by selecting **Build»Target Settings**. Executables created using the **Create Release Executable** command do not contain any debugging information and therefore cannot be debugged.

To create an executable that you can debug, select **Build»Configuration»Debug** and then select **Build»Create Debuggable Executable**.

Creating DLLs in LabWindows/CVI

In LabWindows/CVI, 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 that are compatible with both external compilers.

You must have a separate project for each DLL you want to create. Select **Build»Target Type»Dynamic Link Library** in the Workspace window. Next, you must select **Build»Configuration»Release**. After you have done this, the **Create Release Dynamic Link Library** command appears in the **Build** menu. Use the **Create Release Dynamic Link Library** command to create a DLL suitable for distribution. You can set the name of the DLL as well as other options by selecting **Build»Target Settings**. DLLs created using the **Create Release Dynamic Link Library** command do not contain any debugging information and therefore cannot be debugged.

To create a DLL that you can debug, select **Build»Configuration»Debug** and then select **Build»Create Debuggable Dynamic Link Library**.

Customizing an Import Library

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

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

After you generate the import source, you can modify it, including making calls to functions in other source files. Create a new project that contains the import source file and any other files it refers to. Select **Build»Target Type»Static Library** in the Workspace window. Execute the **Build»Create Static Library** command.

Note This 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 code that LabWindows/CVI generates does not load the DLL until you call one of the functions it exports.

Preparing Source Code for Use in a DLL

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

- The calling convention you use to declare the functions you want to export
- How you specify which DLL functions and variables you want to export
- How you mark imported symbols 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. This section also discusses these differences.

Some of the information in this section is very technical and complex. This section also includes 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 only C or C++ programs to use your DLL, you can use the `__cdecl` convention to declare the functions you want to export. However, if you want your DLL to be callable from environments such as Microsoft Visual Basic, you must declare the functions you want to export with the `__stdcall` calling convention. You must do this by explicitly defining the functions with the `__stdcall` keyword, regardless of whether or not you 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.

Other platforms, such as UNIX or Windows 3.1, do not recognize the __stdcall keyword. If you work with source code that you might use on other platforms, you must use a macro in place of __stdcall. The cvidef.h include file defines the DLLSTDCALL macro for this purpose.

The DLLSTDCALL macro is used in the following examples:

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

Note You cannot use the __stdcall calling convention on functions with a variable number of arguments. Consequently, you cannot use such functions in Microsoft Visual Basic.

Exporting DLL Functions and Variables

When a program uses a DLL, it can access only the functions or variables that the DLL exports. The DLL can export only globally declared functions and variables. The DLL cannot export functions and variables you declare as static.

If you create your DLL in LabWindows/CVI, you can indicate which functions and variables to export in three ways: using the include file method, using the export qualifier method, and using both the include file and export qualifier methods.

Using the Include File Method

You can use include files to identify symbols to export. The include files must contain the declarations of the symbols you want to export. The include files can contain nested #include statements, but the DLL does not export the declarations in the nested include files. In the Target Settings dialog box, select from a list of all the include files in the project.

The include file method does not work with other compilers. However, it is similar to the .def method that the other compilers use.

Using the Export Qualifier Method

You can mark each function and variable you want to export with an export qualifier. Currently, not all compilers recognize the same export qualifier names. The most commonly used qualifier is __declspec(dllexport). Some compilers also recognize __export. LabWindows/CVI recognizes both. The cvidef.h include file defines the DLLEXPORT macro to resolve differences among compilers and platforms. The DLLEXPORT macro is used in the following examples:

```
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 example:

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

Note Borland C/C++ version 4.5*x* requires that you place the qualifier before the asterisk. In Borland C/C++ 5.0 and Borland C++ Builder, 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 your DLL project includes an object or library file that defines exported symbols, LabWindows/CVI cannot correctly create import libraries for each of the external compilers. This problem does not arise if you use only source code files in your DLL project.

Using Both the Include File and Export Qualifier Methods

You can export symbols declared in a header file, as well as symbols marked for export with an export qualifier, by selecting the **Include File and Marked Symbols** option in the DLL Export Options dialog box, which you can access by selecting **Change** in the Target Settings dialog box.

Marking Imported Symbols in an Include File Distributed with a DLL

If your DLL might be used in a C or C++ environment, you must distribute an include file with your DLL. The include file must declare all the symbols the DLL exports. If any of these symbols are variables, you must mark them with an import qualifier. Variable declarations require import qualifiers so the compiler can generate the correct code for accessing the variables.

You can use import qualifiers on function declarations, but they are not necessary. 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 the following 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 you define the variable, LabWindows/CVI and the external compilers treat the variable as if it were imported from *another* DLL and generate 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 the following ways:

- You can avoid exporting variables from DLLs, thereby eliminating 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 and works in LabWindows/CVI. However, this method does not work if you use an export qualifier in a function definition and you create the DLL with an external compiler that requires the declaration to use the same qualifier.
- 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 in the Compiler Defines section of the Build Options dialog box.

Recommendations for Creating a DLL

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

- Use the `DLLSTDCALL` macro in the declaration and definition of all functions you want to export. Do not export functions with a variable number of arguments.
- Identify the symbols you want to export using the include file method. Do not use export qualifiers. If you use an external compiler, use the `.def` file method.
- Do not export variables from the DLL. For each variable you want to export, create functions to get and set the variable 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 gain the following benefits:

- You can distribute with your DLL the same include file that you include in the source files you use to make the DLL. This is especially useful when you create DLLs from instrument drivers.
- You can use the same source code to create the DLL in LabWindows/CVI and both 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

In the Target Settings dialog box, you can 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 that contains `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 Microsoft Windows help file from it using the **Options»Generate Windows Help** command in the Function Tree Editor. The Type Library dialog box, which you can access by clicking the **Type Library** button in the Target Settings dialog box, provides an option to include links to the Windows help file in the Type Library. These links allow Visual Basic users to access the help information from the Type Library Browser.

Visual Basic has a more restricted set of types than C. Also, the **Create Release Dynamic Link Library** and **Create Debuggable Dynamic Link Library** commands impose certain requirements on the declaration of the DLL API. Use the following guidelines to ensure that Visual Basic can use your DLL.

- 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. 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 that work with both compatible external compilers but only if you do not include object or library files in the project.

You must have a separate project for each static library you want to create. Select **Build» Target Type»Static Library** in the Workspace window. Next, select **Build»Configuration» Release**. After you have done this, the **Create Static Library** command appears in the **Build** menu. Use the **Create Static Library** command to create a static library. You can set the name of the static library, as well as other options, by selecting **Build»Target Settings**. Static libraries created using the **Create Static Library** command do not contain any debugging information and therefore cannot be debugged.

Note If you include a `.lib` file in a static library project, LabWindows/CVI includes all object modules from the `.lib` in the static library it creates. When you create an executable or DLL, LabWindows/CVI uses only the necessary modules from the `.lib` file.

Note Do *not* set the default calling convention to `__stdcall` if you want to create a static library for both compatible external compilers.

Creating Object Files

To create an object file in LabWindows/CVI, open a source (`.c`) file and select **Options» Create Object File**.

In LabWindows/CVI, you can choose to create an object file for only the currently selected compiler or to create object files for both compatible external compilers.

Note Do *not* set the default calling convention to `__stdcall` if you want to create an object file for both compatible external compilers.

Note LabWindows/CVI automatically creates object files each time it compiles a source file in the project. These object files have either a `.niobj` or `.nidobj` file extension, depending on whether they are compiled with debugging. LabWindows/CVI uses these files to speed up the building of executables, DLLs, and static libraries. The files cannot be added to a LabWindows/CVI project, used in an external compiler, or loaded using the `LoadExternalModule` function in the Utility Library.

Calling Windows SDK Functions in LabWindows/CVI

You can call Windows SDK functions in LabWindows/CVI. From the LabWindows/CVI Full Development System installation, you can select to install the Windows SDK. This option allows you to call all the Windows SDK functions. Otherwise, you can call only a subset of the Windows SDK functions.

To view help for the SDK functions, select **Help»Windows SDK**.

Windows SDK Include Files

You must include the SDK include files *before* the LabWindows/CVI include files. This way, you avoid problems that arise from 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, LabWindows/CVI must process the SDK include files first, followed by the LabWindows/CVI include files.

When you compile in LabWindows/CVI or when you use an external compiler to compile your source files for linking in LabWindows/CVI, use the LabWindows/CVI SDK include files. The LabWindows/CVI SDK include files are in the `CVI70\sdk\include` directory.

The LabWindows/CVI compiler automatically searches the `CVI70\sdk\include` directory. You do not have to add it to your include paths.

When you use an external compiler to compile and link your source files, you must 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 the LabWindows/CVI SDK include files.

The number of SDK include files is very large. Normally, you must include only `windows.h` because it includes many, but not all, of the other include files. The inclusion of `windows.h`, along with its subsidiary include files, significantly increases compilation time and memory usage. `WIN32_LEAN_AND_MEAN` is a macro from Microsoft that speeds compiling by eliminating 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 alter this setting in the Compiler Defines section of the Build Options dialog box.

Using Windows SDK Functions for User Interface Capabilities

The LabWindows/CVI User Interface Library uses the Windows SDK. The User Interface Library is not designed to be used in 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, National Instruments recommends that you base your user interface either entirely on the LabWindows/CVI User Interface Library or entirely on another user interface development system.

Automatic Loading of SDK Import Libraries

All the SDK functions are in DLLs. LabWindows/CVI and the external compilers each come with a number of DLL import libraries for the SDK functions. Most of the commonly used SDK functions are in the following import libraries:

```
kernel32.lib
gdi32.lib
user32.lib
advapi32.lib
uuid.lib
winmm.lib
```

LabWindows/CVI automatically loads these libraries when it starts up and searches them to resolve references at link time. Thus, you do not have to include these libraries in your project.

If the LabWindows/CVI linker reports SDK functions as unresolved references, you must add import libraries to your project. Refer to the SDK help for a function to determine the import library that you need to add to your project. The import libraries are in the `CVI70\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 from the rules for using the LabWindows/CVI library include files.

You must set up your include paths differently depending on the environment in which you compile and link. This manual includes a discussion of each of the following compiling and linking environments:

- Compiling in LabWindows/CVI for Linking in LabWindows/CVI
- Compiling in LabWindows/CVI for Linking in an External Compiler
- Compiling in an External Compiler for Linking in an External Compiler
- Compiling in an External Compiler for Linking in LabWindows/CVI

Compiling in LabWindows/CVI for Linking in LabWindows/CVI

Use the LabWindows/CVI SDK and ANSI C include files. You do not need to set up any special include paths. LabWindows/CVI finds the correct include files automatically.

Compiling in LabWindows/CVI for Linking in an External Compiler

Use the LabWindows/CVI SDK include files and the ANSI C include files from the external compiler. Use the Include Paths dialog box, accessible through the Environment dialog box, to add the following explicit include paths at the beginning of the project-specific list.

```
CVI70\include
CVI70\sdk\include
directory that contains 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 from the external compiler. This action happens automatically. Specify the following directory as an include path in the external compiler for the LabWindows/CVI library include files.

```
CVI70\include
```

Compiling in an External Compiler for Linking in LabWindows/CVI

Use the LabWindows/CVI ANSI C include files. Specify the following directories as include paths in the external compiler.

```
CVI70\include
CVI70\include\ansi
```

Handling Hardware Interrupts under Windows 2000/NT/XP/Me/98

Under Windows Me/98, you must handle hardware interrupts in a VxD. Under Windows 2000/NT/XP, you must handle hardware interrupts in a kernel-mode driver. You cannot create VxDs and kernel-mode drivers in LabWindows/CVI. Instead, you must create them in Microsoft Visual C/C++, and you also must have the Microsoft Driver Developer Kit (DDK).

Under Windows 2000/NT/XP/Me/98, you can arrange for the VxD or kernel-mode driver to call a function in your LabWindows/CVI source code after the interrupt service routine exits. You do this by creating a separate thread for your interrupt callback function. The callback function executes a loop that blocks its thread until the interrupt service routine signals it. 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. LabWindows/CVI also includes a sample main program to show you how to read and write registers on a board. There is one set of files for Windows Me/98 and another for Windows 2000/NT/XP.

The source code template files are in `CVI70\vxd\win95` and `CVI70\vxd\winnt`. The `template.doc` file in each directory contains basic information about the template files.

4

Creating and Distributing Release Executables and DLLs

Introduction to the Run-Time Engine

With your purchase of LabWindows/CVI, you received the Run-time Engine as part of your distribution. The LabWindows/CVI Run-time Engine is necessary to run executables or use DLLs you create with LabWindows/CVI, and it must be present on any target computer on which you want to run your executable program. You can distribute the Run-time Engine according to your license agreement.

Distributing Stand-Alone Executables

You can bundle the LabWindows/CVI Run-time Engine with your distribution kit using the **Build»Create Distribution Kit** command in the Workspace window.

Minimum System Requirements

To use a stand-alone executable or DLL that depends on the LabWindows/CVI Run-time Engine, you must have the following:

- A personal computer using a Pentium 600 or higher microprocessor
- Windows 2000/NT SP6/XP/Me/98
- 800 × 600 resolution (or higher) video adapter
- Minimum of 128 MB of RAM; 256 MB recommended
- Free hard disk space equal to 4 MB, plus space to accommodate your executable or DLL and any files the executable or DLL requires

Note You will need additional hard drive space and memory if the stand-alone executable or DLL uses the DataSocket Library, NIReports instrument driver, or ActiveX controls.

Translating the Message File

The message file, called `msgrtn.txt` where *n* is the version number of the Run-time Engine, is a text file that contains the error messages that the Run-time Engine displays. `msgrtn.txt` resides in the `bin` directory of the Run-time Engine installation directory. To translate the message file into other languages, complete the following steps:

1. Copy the file to another name so you have it as a backup.
2. Use a text editor to modify `msgrte.txt`. Translate only the text that is inside quotation marks. Do not add or delete any message numbers.
3. Execute the `countmsg.exe` or `countmsg` utility on the file to encode it for use with the Run-time Engine, as is shown in the following example:

```
countmsg msgrte.txt msgrte.new
del msgrte.txt
ren msgrte.new msgrte.txt
```

Configuring the Run-Time Engine

This section applies to you, the developer, and the user of your executable program. You may use the text in this section in the documentation for your executable program.

Configuration Option Descriptions

The Run-time Engine recognizes various configuration options. The installation program for the Run-time Engine automatically sets the required configuration options for you.

You also can manually set the configuration options for the LabWindows/CVI development environment. You can set the Run-time Engine configuration options in a similar manner, but you must set the configuration options in the Registry under the following key:

```
HKEY_LOCAL_MACHINE\Software\National Instruments
\CVI Run-Time Engine\cvirte
```

cvidir

`cvidir` specifies the location of the directory that contains the `bin` and `fonts` subdirectories that the Run-time Engine requires. You need to enter this registry entry only if the Run-time Engine `bin` and `fonts` directories are not under the Windows system directory and not under your application's directory.

useDefaultTimer

The LabWindows/CVI Run-time Engine recognizes the `useDefaultTimer` option. This option has the same effect in the LabWindows/CVI Run-time Engine as it has in the LabWindows/CVI development environment.

DSTRules

The LabWindows/CVI Run-time Engine recognizes the `DSTRules` option. This option has the same effect in the LabWindows/CVI Run-time Engine as it has in the LabWindows/CVI development environment.

Necessary Files for Running Executable Programs

In order for your executable to run successfully on a target computer, all files the executable requires must be accessible. Your final distribution kit must contain all of the following files to install your LabWindows/CVI executable program on a target machine.

- **Executable**—File that contains a precompiled, prelinked version of your LabWindows/CVI project and any instrument driver program files that you link to your project. The executable also contains the application name and icon resource to register to the operating system. The executable has an associated icon you can double-click to start the application. When the executable starts, it loads the Run-time Engine.
- **Run-time Engine**—Run-time Engine that contains all the built-in LabWindows/CVI libraries. The Run-time Engine consists of multiple files, including various DLLs and other support files.
- **Hardware Drivers**—Device drivers such as NI-DAQ and NI-VISA. You must install the hardware drivers if your project includes them.
- **Hardware Configurations**—If your project uses MAX-based DAQmx tasks, you must include them. These tasks are loaded using `DAQmxLoadTask`. In the Create Distribution Kit dialog box, enable the **Include Hardware Configuration** option and select a hardware configuration file.
- **NIReports Server**—Automation server that implements the report generation functionality provided by the NIReports instrument driver. Use the Create Distribution Kit **Install NI-Report Support** option to include this installation in a distribution kit.
- **DataSocket Server and Utilities**—Server and its associated utilities that are required if your program uses the functions in the LabWindows/CVI DataSocket Library. Use the Create Distribution Kit feature in LabWindows/CVI to include this installation in a distribution kit.
- **.uir files**—The User Interface Resource files that your application program uses. Use `LoadPanel`, `LoadPanelEx`, and `LoadMenuBar` to load these files. You do not need to include `.uir` files if you have enabled the **Embed Project .UIRs** option in the Target Settings dialog box.
- **Image files**—The graphical image files that you programmatically load and display on your user interface.
- **State files**—The user interface panel state files that you save using `SavePanelState` and load using `RecallPanelState`.

- **DLL files**—The Windows Dynamic Link Library files that your application program uses.
- **External .lib files**—Compiled 32-bit `.lib` files that you load using `LoadExternalModule` and that you have not listed in the project.
- **External .obj files**—Compiled 32-bit `.obj` files that you load using `LoadExternalModule` and that you have not listed in the project.
- **Other files**—Files your executable opens using `open`, `fopen`, `OpenFile`, and so on.
- **ActiveX components**—Any external ActiveX components that you use need to be included.
- **ActiveX container**—In the Create Distribution Kit dialog box, enable the **Install ActiveX Container Support** option if you use ActiveX controls in your `.uir` files.

Necessary Files for Using DLLs

You can distribute DLLs that use the LabWindows/CVI Run-time Engine. As in the case of stand-alone executables, you must distribute them along with the LabWindows/CVI Run-time Engine.

Location of Files on the Target Machine for Running Executables and DLLs

To assure proper execution, it is critical that all files associated with your executable program are in the proper directories on the target machine. Specify these files in a relative directory structure in the Create Distribution Kit dialog box.

LabWindows/CVI Run-Time Engine

Table 4-1 shows the files that comprise the LabWindows/CVI Run-time Engine.

Table 4-1. Run-Time Engine Files

Run-Time Engine File	Description
`cvirte.dll`	Contains most LabWindows/CVI libraries
`cviauto.dll`	Contains ActiveX Library
`cvi95vxd.vxd`	Low-level support driver for Windows Me/98
`cvintdrv.sys`	Low-level support driver for Windows 2000/NT/XP
`msgrte.txt`	Contains text messages

Table 4-1. Run-Time Engine Files (Continued)

Run-Time Engine File	Description
cvirte.rsc	Contains binary resources
ni7seg.ttf	Font description file
nisystem.ttf	Font description file
dataskt.dll	LabWindows/CVI support DLL for the DataSocket Library
mesa.dll	LabWindows/CVI support DLL for Lab-Style controls

The LabWindows/CVI installation program installs the files as part of the development environment. Use the **Build»Create Distribution Kit** command in the Workspace window to bundle the Run-time Engine DLLs and drivers into your distribution kit.

Run-Time Library DLLs

The LabWindows/CVI installation program, the Run-time Engine installation program, and the Create Distribution Kit installation programs place the Run-time Engine DLLs in the Windows `system` directory under Windows Me/98 and the Windows `system32` directory under Windows 2000/NT/XP.

Use the Create Distribution Kit feature to create an installation program that will install the Run-time Engine DLLs in your application's directory. If you choose to install the Run-time Engine DLLs in your application directory, you also must install the message, resource, and font files in subdirectories of your application directory. The application directory should follow the following structure:

```
UserAppDir/userapp.exe
    /userapp.uir
    /cvirte.dll
    /cvirt.dll
    /cvirte
        /bin
            /msgrte.txt
            /cvirte.rsc
        /fonts
            /ni7seg.ttf
            /nisystem.ttf
```

Low-Level Support Driver

The Run-time Engine loads the low-level support driver if it is present when you start your stand-alone executable. Several functions, including `CVILowLevelSupportDriverLoaded`, in the Utility Library require the low-level support driver.

The installation program installs the low-level support driver in the Windows `system` directory under Windows Me/98 and the Windows `system32\drivers` directory under Windows 2000/NT/XP. Under Windows 2000/NT/XP, the installation program also adds a registry entry under the following key:

`HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\cvintdrv`

Table 4-2 shows the values the installation program sets for the Windows NT registry entry for the low-level support driver.

Table 4-2. Windows NT Registry Entry Values

Type	Name	Value
DWORD	`ErrorControl`	`00000001`
String	`Group`	`"Extended Base"`
DWORD	`Start`	`00000002`
DWORD	`Type`	`00000001`

Message, Resource, and Font Files

The installation program installs `ni7seg.ttf` and `nisystem.ttf` in the `cvirte\fonts` subdirectory under the directory in which it installs the Run-time Engine DLLs. The installation program installs `msgrte.txt` and `cvirte.rsc` in the `cvirte\bin` subdirectory under the directory in which it installs the Run-time Engine DLLs.

If the Run-time Engine DLLs are installed in the Windows `system` or `system32` directories, you can subsequently change the location of the `bin` and `fonts` subdirectories, but you also must set the `cvidir` registry option to the pathname of the directory that contains the two subdirectories.

National Instruments Hardware I/O Libraries

The LabWindows/CVI Run-time Engine does not include the DLLs or drivers for National Instruments hardware. Users can install the DLLs and drivers for their hardware from the distribution disks that National Instruments supplies.

Rules for Accessing .uir, Image, and Panel State Files

To access `.uir`, image, and panel state files in your executable program, place the files in the same directory as the executable and pass simple filenames with no drive letters or directory names to `LoadPanel`, `LoadPanelEx`, `DisplayImageFile`, `SavePanelState`, and `RecallPanelState`.

If you do not want to store these files in the same directory as your executable, you must pass pathnames to `LoadPanel`, `LoadPanelEx`, `DisplayImageFile`, `SavePanelState`, and `RecallPanelState`. These functions interpret relative pathnames as being relative to the directory that contains the executable.

Note You do not need to include `.uir` files if you have enabled the **Embed Project .UIRs** option in the Target Settings dialog box.

Rules for Using DLL Files

Your executable or DLL can link to a DLL only through an import library. This section refers to a DLL that an executable or another DLL uses as a *subsidiary* DLL. You can link an import library into your program in any of the following ways:

- List the DLL import library in your project
- Associate the DLL import library with the `.fp` file for an instrument driver or user library
- Dynamically load the DLL import library by calling `LoadExternalModule`

If you list a DLL import library in the project or associate it with an instrument driver or user library, LabWindows/CVI statically links the import library into your executable or DLL. However, if you load the import library through a call to `LoadExternalModule`, you must distribute it separately from your executable.

Regardless of the method you use to link the import library, you must distribute the subsidiary DLL separately. The import library always contains the name of the subsidiary DLL. When your executable or DLL is loaded, the operating system finds the subsidiary DLL using the standard DLL search algorithm, which the Windows SDK documentation for the `LoadLibrary` function describes. The search precedence is as follows:

1. The directory from which the user loads the application
2. The current working directory
3. Under Windows Me/98, the Windows `system` directory. Under Windows 2000/NT/XP, the Windows `system32` and `system` directories
4. The Windows directory
5. The directories listed in the `PATH` environment variable

The **Create Distribution Kit** command automatically includes in your distribution kit the DLLs that the import libraries in your project refer to. You must add to the distribution kit any DLLs that you load through `LoadExternalModule` or that you load by calling the Windows SDK `LoadLibrary` function.

Do not include DLLs for National Instruments hardware in your distribution kit. The user must install these DLLs from the distribution disks that National Instruments supplies.

Rules for Loading Files Using LoadExternalModule

`LoadExternalModule` can load the file types shown in Table 4-3.

Table 4-3. Loadable File Types

File Type	File Extension
Library Files	`.lib`
Object Modules	`.obj`
DLL Import Library Files	`.lib`
Source Files	`.c` (linked into your executable or DLL)

Forcing Modules that External Modules Refer to into Your Executable or DLL

LabWindows/CVI includes in the executable only modules that your project refers to directly. If an external module refers to modules not included in the executable, calls to `RunExternalModule` or `GetExternalModuleAddr` on that external module fail.

To avoid this problem, you must force any missing modules into your executable or DLL. Select **Add Files to Executable** or **Add Files to DLL** in the Target Settings dialog box to open a list of project `.lib`, instrument, and library files. Select the files you want to include in your executable or DLL. If you select a `.lib` file, it is linked in its entirety.

Alternatively, you can link modules into your executable or DLL by including dummy references to them in your program. For instance, if your external module references the functions `FuncX` and `FuncY`, include the following statement in your program

```
void *dummyRefs[] = {(void *)FuncX, (void *)FuncY};
```

Using LoadExternalModule on Files in the Project

You can call `LoadExternalModule` on files listed in the project. You must pass the simple filename to `LoadExternalModule`. However, when you create an executable or DLL from your project, you might have additional work to do.

- If you *link your executable or DLL in LabWindows/CVI*, the following rules apply for files listed in the project.
  - For `.c` or `.obj` files, everything works automatically.
  - For `.lib` files, by default, the following commands link in only the library modules that you reference statically in the project: **Create Debuggable Executable**, **Create Release Executable**, **Create Debuggable Dynamic Link Library**, or **Create Release Dynamic Link Library**. Therefore, you must force into the executable the modules that contain the functions you call using `GetExternalModuleAddr`.

    To force these modules into the executable, include the library file in the project and take one of the following actions:

    - If you want to force the entire library file into the executable, click **Add Files to Executable** or **Add Files to DLL** in the Target Settings dialog box. The **Add Files to Executable** button appears in the Target Settings dialog box when you set **Target Type** to **Executable**. The **Add Files to DLL** button appears when you set **Target Type** to **Dynamic Link Library**.
    - If you want to force only specific modules from the library into the executable, reference them statically in your program. For example, you could have an array of void pointers and initialize them to the names of the necessary symbols.

Note Import libraries may contain functions that are not in the corresponding DLL. For example, the Windows SDK import libraries contain some functions that are not present on Windows 2000/NT/XP/Me/98. For this reason, your program may not run on one or more of these platforms if you force a Windows SDK import library into your program using **Add Files to DLL** or **Add Files to Executable**.

- If you *link in an external compiler*, the LabWindows/CVI Utility Library does not know the location of symbols in the externally linked executable or DLL. Consequently, without further action on your part, you cannot call `GetExternalModuleAddr` or `RunExternalModule` on modules that you link directly into your executable or DLL. You have the following alternatives:
  - Remove the file from the project and distribute it as a separate `.obj`, `.lib`, or `.dll`.
  - Use the Other Symbols section of the External Compiler Support dialog box, accessible through in the **Build** menu of the Workspace window, to create an object module that contains a table of symbols you want `GetExternalModuleAddr` to find. If you use this method, pass the empty string (`""`) to `LoadExternalModule`

as the module pathname. The empty string indicates that you linked the module directly into your executable or DLL using an external compiler.

Using LoadExternalModule on Library and Object Files Not in the Project

If you call `LoadExternalModule` on a library or object file not in the project, you must keep the library or object file separate in your distribution.

When you keep an object or library file separate, you can manage memory more efficiently and replace the file without having to replace the executable. For this reason, if you call `LoadExternalModule` on a library or object file in the project, remove or exclude the file from the project before you select **Create Debuggable Executable**, **Create Release Executable**, **Create Debuggable Dynamic Link Library**, or **Create Release Dynamic Link Library**, and then include it as a separate file when you use **Create Distribution Kit**.

However, remember that you cannot statically reference functions defined in a separate library or object file from the executable or DLL. You must use `LoadExternalModule` and `GetExternalModuleAddr` to make such references.

When you distribute the library or object file as a separate file, it is a good idea to place the file in the same directory as the executable or DLL. If you place the file in the same directory, you can pass a simple filename to `LoadExternalModule`. If you do not want the file to be in the same directory as your executable, you must pass a pathname to `LoadExternalModule`. `LoadExternalModule` interprets relative pathnames as being relative to the directory that contains the executable or DLL.

Using LoadExternalModule on DLL Files

You cannot pass the pathname of a DLL directly into `LoadExternalModule`. Instead, you must pass the pathname of a DLL import library. You can link the import library into your executable or DLL or distribute it separately and load it dynamically.

You always must distribute DLLs as separate files. The operating system finds the DLL associated with the loaded import library using the standard Windows DLL search algorithm. The search precedence is as follows:

1. The directory from which the user loads the application
2. The current working directory
3. Under Windows Me/98, the Windows `system` directory. Under Windows 2000/NT/XP, the Windows `system32` and `system` directories
4. The Windows directory
5. The directories listed in the `PATH` environment variable

Using LoadExternalModule on Source Files (.c)

If you pass the name of a source file to `LoadExternalModule`, the source file must be in the project. LabWindows/CVI automatically compiles the source file and links it into the executable when you select **Create Debuggable Executable**, **Create Release Executable**, **Create Debuggable Dynamic Link Library**, or **Create Release Dynamic Link Library**. For this reason you must pass a simple filename to `LoadExternalModule`. If you use an external compiler, refer to the *Using LoadExternalModule on Files in the Project* section.

If the source file is an instrument driver program that is not in the project and you link in LabWindows/CVI, you have two alternatives.

- Add the instrument driver `.c` source to the project.
- Refer to one of the variables or functions it exports in one of your project files.

If the source file is an instrument driver program that is not in the project and you link in an external compiler, you must create an object file and keep it separate from the executable.

Rules for Accessing Other Files

The functions for accessing files, such as `fopen`, `OpenFile`, `SetFileAttrs`, `DeleteFile`, and so on, interpret relative pathnames as being relative to the current working directory. The initial current working directory is normally the directory of the executable. However, if a different directory exists in the **Working Directory** or **Start In** field of the Windows Properties dialog box for the shortcut for the executable, then that directory is the initial current working directory. You can create an absolute path for a file in the executable directory by using `GetProjectDir` and `MakePathname`.

Error Checking in Your Release Executable or DLL

Usually, you enable debugging and the **Break on»Library Errors** option while you develop your application in LabWindows/CVI. With these features enabled, LabWindows/CVI checks for programming errors in your source code. Consequently, you might have a tendency to relax your own error checking.

When you create a release executable program or DLL, debugging and the **Break on» Library Errors** option are disabled, resulting in smaller and faster code. Thus, you must perform your own error checking when you create a release executable program or DLL.

5

Distributing Libraries and Function Panels

Distributing Libraries

You can distribute libraries for other users to include in their Library Trees. You must create a function panel (`.fp`) for each library program file. If you do not want to develop function panels for the library functions, create a `.fp` file without any classes or functions. In that case, LabWindows/CVI loads the library at startup but does not include the library name in the Library Tree. This approach is useful when the library supports other libraries and contains no user-callable functions.

Adding Libraries to a User's Library Tree

Normally, a user must manually add libraries to the Library Tree using the **Library» Customize** command in the Workspace window. However, you can insert your libraries into the user's Library Tree by modifying the user's Registry.

The `modreg` program is in the LabWindows/CVI `bin` subdirectory for this purpose. A documentation file called `modreg.doc` and the source code are in the same directory.

Assume that you install function panels for two libraries in the following location:

```
c:\newlib\lib1.fp
c:\newlib\lib2.fp
```

To add the libraries to the user's Library Tree, use the following `modreg` command file:

```
setkey [HKEY_LOCAL_MACHINE\Software\National Instruments]
appendkey CVI70\@latestVersion
add AutoloadLibraries LibraryFPFile "c:\newlib\lib1.fp"
add AutoloadLibraries LibraryFPFile "c:\newlib\lib2.fp"
```

After the user installs the library files, the `modreg` program must be run on the user's disk using the command file.

Caution Make sure that LabWindows/CVI is not running when you use the `modreg` program to modify the Registry. If LabWindows/CVI is running while you use this program, you will lose your changes.

Specifying Library Dependencies

When one library you distribute depends on another library you distribute, you can specify this dependency in the function panel file for the dependent library. When LabWindows/CVI loads the dependent library, it attempts to load the libraries upon which it depends. Use the **Edit»FP Auto-Load List** command in the Function Tree Editor of the dependent library to list the `.fp` files of the libraries upon which it depends.

LabWindows/CVI can find the required libraries most easily when they are all in the same directory as the dependent library. When you cannot put them in the same directory, you must add the directories in which the required libraries reside to the user's Instrument Directories list. The user can manually enter this information using the **Instrument»Search Directories** command in the Workspace window. Also, you can add to the Instrument Directories list by editing the Registry.

The `modreg` program is in the LabWindows/CVI `bin` subdirectory for this purpose. A documentation file called `modreg.doc` and the source code are in the same directory.

Assume that you install two `.fp` files in the following locations:

```
c:\newlib\liba.fp
c:\genlib\libb.fp
```

If `liba` depends on `libb`, you must add the following path to the user's Instrument Directories list:

```
c:\genlib
```

For LabWindows/CVI to be able to find the dependent file, use the following `modreg` command file:

```
setkey [HKEY_CURRENT_USER\Software\National Instruments]
appendkey CVI70\@latestVersion
add InstrumentDirectories InstrDir "c:\genlib"
```

After the user installs the library files, the `modreg` program must be run on the user's disk using the command file.

Caution Make sure that LabWindows/CVI is not running when you use the `modreg` program to modify the Registry. If LabWindows/CVI is running while you use this program, you will lose your changes.

6

Checking for Errors in LabWindows/CVI

This chapter describes LabWindows/CVI error checking and how LabWindows/CVI reports errors in LabWindows/CVI libraries and compiled external modules.

When you develop applications in LabWindows/CVI, you usually have debugging and the **Break on»Library Errors** option enabled. With these features enabled, LabWindows/CVI identifies and reports programming errors in your source code. Therefore, you might have a tendency to relax your own error checking. However, in compiled modules and release executables, debugging and the **Break on»Library Errors** option are disabled. This results in smaller and faster code, but you must perform your own error checking. This fact is important to remember because many problems can occur in compiled modules and release executables, even if the program works inside the LabWindows/CVI environment.

It is important to check for errors that can occur because of external factors beyond the control of your program. Examples include running out of memory or trying to read from a file that does not exist. `malloc`, `fopen`, and `LoadPanel` are examples of functions that can encounter such errors. You must provide your own error checking for these types of functions. Other functions return errors only if your program is incorrect. The following function call returns an error only if `pnl` or `ctrl` is invalid.

```
SetCtrlAttribute(pnl, ctrl, ATTR_DIMMED, FALSE);
```

The **Break on»Library Errors** feature of LabWindows/CVI adequately checks for these types of errors while you develop your program and external factors do not affect this function call. Therefore, it is generally not necessary to perform explicit error checking on this type of function call.

You can check for errors by checking the status of function calls upon their completion. Most functions in commercial libraries return errors when they encounter problems. LabWindows/CVI libraries are no exception. All the functions in the LabWindows/CVI libraries and in the instrument drivers available from National Instruments return a status code to indicate the success or failure of execution. These codes help you determine the problem when the program does not run as you expected. This section describes how LabWindows/CVI reports these status codes and some techniques for checking them.

Note LabWindows/CVI libraries and National Instruments instrument drivers return status codes that are integer values. These values are either common to an entire library of functions or specific to one function. Each of these libraries contains a function you can call to translate the integer value to an error string. When an error code is specific to a function, you can find a description for it in the *Library Reference* section of the *LabWindows/CVI Help*.

Error Checking

LabWindows/CVI functions return status codes in one of two ways—either by a function return value or by updating a global variable. In some cases, LabWindows/CVI uses both of these methods. In either case, it is a good idea to monitor these values so that you can detect an error and take appropriate action. You can monitor the status of functions, and when a function reports an error, pause the program and report the error to the user through a pop-up message. For example, `LoadPanel` returns a positive integer when it successfully loads a user interface panel into memory. However, if a problem occurs, the return value is negative. The following example shows an error message handler for `LoadPanel`.

```
panelHandle = LoadPanel (0, "main.uir", PANEL);
if (panelHandle < 0) {
    ErrorCheck ("Error Loading Main Panel", panelHandle,
        GetUILErrorString (panelHandle));
}
```

When a function reports status through a separate function, as in the RS-232 Library, check for errors in a similar way. In this case, the status function returns a negative value when the original function fails.

```
bytesRead = ComRd (1, buffer, 10);
if (ReturnRS232Error() < 0) {
    ErrorCheck ("Error Reading From ComPort #1", ReturnRS232Error(),
        GetRS232ErrorString(ReturnRS232Error()));
}
```

Notice that the preceding function also returns the number of bytes read from the serial port. You can compare the number of bytes read to the number you request, and if a discrepancy exists, take the appropriate action. Notice that the error codes differ between the RS-232 Library and the User Interface Library. The *Status Reporting by LabWindows/CVI Libraries and Instrument Drivers* section describes how each LabWindows/CVI library reports errors.

After your program detects an error, it must take some action to either correct the situation or prompt the user to select a course of action. The following example shows a simple error response function.

```
void ErrorCheck (char *errMsg, int errVal, char *errString)
{
```

```
    char outputMsg[256];
    int response;
    Fmt (outputMsg, "%s (Error = %d).\n%s\nContinue? ", errMsg, errVal,
        errString);
    response = ConfirmPopup ("ErrorCheck", outputMsg);
    if (response == 0)
        exit (-1);
}
```

Status Reporting by LabWindows/CVI Libraries and Instrument Drivers

This section describes how LabWindows/CVI libraries and instrument drivers report errors. Notice that libraries that return their status code using global variables or separate functions sometimes report additional status information through return values.

Status Reporting by the User Interface Library

The User Interface Library routines return negative values when they detect errors. Some functions, such as `LoadPanel`, return positive values for a successful completion. You can use the `GetUILErrorString` function to get the error message associated with each User Interface Library error code.

Status Reporting by the Analysis and Advanced Analysis Libraries

The Analysis and Advanced Analysis Library functions return negative values when they detect errors. The function panel help contains error codes. You can use the `GetAnalysisErrorString` function to get the error message associated with each Analysis Library error code.

Status Reporting by the Traditional NI-DAQ Library

The Traditional NI-DAQ Library functions return negative values when they detect errors. They return positive values as warnings when they are able to complete their tasks but not in the way you might expect. This library uses a common set of error codes. The positive warning codes are the same absolute values as the negative error codes.

You can use `GetNIDAQErrorString` to get the error message associated with each Traditional NI-DAQ Library error or warning code.

Status Reporting by the NI-DAQmx Library

The NI-DAQmx Library functions return negative values when they detect errors. They return positive values as warnings when they are able to complete their tasks but not in the way you might expect.

You can use `DAQmxGetErrorString` to get the error message associated with each NI-DAQmx Library error or warning code. Use `DAQmxGetExtendedErrorInfo` to get additional information about the last error reported. For example, if an error occurs in an attribute function and you call `DAQmxGetExtendedErrorInfo`, LabWindows/CVI provides information such as the attribute name, what you set the attribute to, and what values are valid for that attribute in addition to the error message.

Status Reporting by the VXI Library

The VXI Library uses a variety of global variables and function return codes to report any error that occurs. You must check each function description to determine what error checking might be necessary. Refer to the *NI-VISA Programmer Reference Help* for a list of the error codes or the function panel help for descriptions of the error codes.

Status Reporting by the GPIB/GPIB 488.2 Library

The GPIB libraries return status information through two global variables called `ibsta` and `iberr`.

Note If your program uses multiple threads, use the `ThreadIbsta` and `ThreadIberr` functions in place of the `ibsta` and `iberr` global variables.

Note The GPIB/GPIB 488.2 Library functions return the same value that they assign to `ibsta`. You can choose to use the return values, `ibsta`, or `ThreadIbsta`.

The `ERR` bit within `ibsta` indicates an error condition. If this bit is not set, `iberr` does not contain meaningful information. If the `ERR` bit is set in `ibsta`, the error condition is stored in `iberr`. After each GPIB call, your program should check whether the `ERR` bit is set to determine if an error has occurred, as shown in the following code segment:

```
if (ibwrt(bd[instrID], buf, cnt) & ERR)
    PREFIX_err = 230;
```

Refer to the *NI-488.2 Help* for detailed information about GPIB global variables and lists of status and error codes. LabWindows/CVI function panel help also has lists of status and error codes.

Status Reporting by the RS-232 Library

The RS-232 Library returns status information through a global variable called `rs232err`. If this variable is negative after the function returns, an error occurred. Notice that many of the functions return a value in addition to setting the global variable. Usually, this value contains information about the result of the function that also can be used to detect a problem. Each function should be checked individually. You can use `GetRS232ErrorString` to get the error message associated with each RS-232 Library error code.

Note If your program uses multiple threads, use the `ReturnRS232Err` function in place of the `rs232err` global variable.

Status Reporting by the VISA Library

The VISA Library functions return negative values when they detect errors. The functions return positive values as alternate success codes or as warnings. This library uses a common set of error and warning codes, but the warning code values are entirely separate from the error code values. The error codes always contain 0xBFFF in the upper two bytes. The warning codes always contain 0x3FFF in the upper two bytes. Refer to the *NI-VISA Programmer Reference Help* or the function panel help for a list of the error and warning codes and information on the individual functions. You can use `viStatusDesc` to obtain the error message associated with each VISA Library error code.

Status Reporting by the IVI Library

The IVI Library functions return negative values when they detect errors. This library uses a common set of error codes. Refer to the *Library Reference»IVI Library Help* section of the *LabWindows/CVI Help* or the function panel help for a list of the error codes and information on the individual functions. IVI Library functions sometimes also provide a secondary error code or an elaboration string to give you additional information about an error condition. You can use `Ivi_GetErrorInfo` to obtain the primary error code, secondary error code, and the elaboration string. You can use `Ivi_GetErrorMessage` to obtain the error message associated with each IVI Library error code.

Status Reporting by the TCP Support Library

The TCP Support Library functions return negative values when they detect errors. This library uses a common set of error codes. You can use `GetTCPErrorString` to get the error message associated with each TCP Support Library error code.

Status Reporting by the DataSocket Library

The DataSocket Library functions return negative values when they detect errors. This library uses a common set of error codes.

Status Reporting by the DDE Support Library

The DDE Support Library functions return negative values when they detect errors. This library uses a common set of error codes. Refer to the function panel help for a list of the error codes and information on the individual functions. You can use the `GetDDEErrorString` function to get the error message associated with each DDE Support Library error code.

Status Reporting by the ActiveX Library

The ActiveX Library functions return negative values when they detect errors. This library uses a common set of error codes. You can use `CA_GetAutomationErrorString` to get the error message associated with each ActiveX Library error code.

Status Reporting by the Formatting and I/O Library

The Formatting and I/O Library functions return negative error codes when they detect errors. However, you must keep in mind an important fact. When you enable debugging, the LabWindows/CVI environment keeps track of the sizes of strings and arrays. If LabWindows/CVI detects any attempt to access a string or array beyond its boundary, the environment halts the program and informs you. It is important to remember that this feature works only when you execute source code in the LabWindows/CVI development environment. The string functions can write beyond the end of a string or array without detection, resulting in corruption of memory. Therefore, you must use the Formatting and I/O functions on strings and arrays with caution.

In addition to the return codes, the `GetFmtErrNdx` and `NumFmtdBytes` functions return information about how the last scanning and formatting function executed. The `GetFmtIOError` function returns a code that contains specific error information about the last Formatting and I/O Library function that performed file I/O. The `GetFmtIOErrorString` function converts this code into an error string.

Status Reporting by the Utility Library

Utility Library functions report error codes as return values. You can check each individual function description in the Utility Library or in the function panel help to determine the error conditions that can occur in each function.

Status Reporting by the ANSI C Library

Some of the ANSI C Library functions report error codes as return values. Some functions also set the global variable `errno`. Generally, the functions do not clear `errno` when they return successfully. To learn more about these values, refer to a publication such as *C: A Reference Manual* cited in the *Related Documentation* section of *About This Manual*. You also can use the LabWindows/CVI function panel help to determine the error conditions that can occur in each function.

Status Reporting by LabWindows/CVI Instrument Drivers

Instrument drivers from National Instruments use a standard status reporting scheme. Functions report error codes as return values, and you can check each function individually in the LabWindows/CVI function panel help to determine the error conditions that can occur in each function.

Instrument drivers that comply with the VXI*plug&play* standard contain two error reporting functions. *Prefix*_error_query, where *Prefix* is the instrument prefix, allows you to query the error queue in the physical instrument. If the instrument does not have an error queue, *Prefix*_error_query returns the VI_WARN_NSUP_ERROR_QUERY warning code from the VISA Library. *Prefix*_error_message translates the error and warning codes that the other instrument driver functions return into descriptive strings.

IVI instrument drivers are VXI*plug&play* compliant and contain the *Prefix*_error_query and *Prefix*_error_message functions. In addition, IVI instrument driver functions sometimes also provide a secondary error code or an elaboration string to give you additional information about an error condition. You can use *Prefix*_GetErrorInfo to obtain the primary error code, secondary error code, and the elaboration string for the first error that occurred on a particular instrument session or in the current thread since you last called *Prefix*_GetErrorInfo. You also can use the *Prefix*_GetAttribute function to obtain each of these data items individually for the most recent function call on a particular instrument session.

A

Technical Support and Professional Services

Visit the following sections of the National Instruments Web site at ni.com for technical support and professional services:

- **Support**—Online technical support resources include the following:
  - **Self-Help Resources**—For immediate answers and solutions, visit our extensive library of technical support resources available in English, Japanese, and Spanish at ni.com/support. These resources are available for most products at no cost to registered users and include software drivers and updates, a KnowledgeBase, product manuals, step-by-step troubleshooting wizards, conformity documentation, example code, tutorials and application notes, instrument drivers, discussion forums, a measurement glossary, and so on.
  - **Assisted Support Options**—Contact NI engineers and other measurement and automation professionals by visiting ni.com/support. Our online system helps you define your question and connects you to the experts by phone, discussion forum, or email.
- **Training**—Visit ni.com/training for self-paced tutorials, videos, and interactive CDs. You also can register for instructor-led, hands-on courses at locations around the world.
- **System Integration**—If you have time constraints, limited in-house technical resources, or other project challenges, NI Alliance Program members can help. To learn more, call your local NI office or visit ni.com/alliance.

If you searched ni.com and could not find the answers you need, contact your local office or NI corporate headquarters. Phone numbers for our worldwide offices are listed at the front of this manual. You also can visit the Worldwide Offices section of ni.com/niglobal to access the branch office Web sites, which provide up-to-date contact information, support phone numbers, email addresses, and current events.

Glossary

A

API	Application Programming Interface.

C

compatible external compiler	The two external 32-bit compilers with which LabWindows/CVI is compatible: Microsoft Visual C/C++ and Borland C/C++.
control	Object that resides on a panel and provides a mechanism for accepting input from and displaying information to the user.
current compatible compiler	The compiler with which your copy of LabWindows/CVI is currently compatible.

D

dialog box	A prompt mechanism in which you specify additional information needed to complete a command.

E

external module	A compiled module you load dynamically from your program.

F

`.fp` file	A file that contains information about the function tree and function panels of an instrument driver.
function panel	A screen-oriented user interface to the LabWindows/CVI libraries that allows interactive execution of library functions and is capable of generating code for inclusion in a program.
function tree	The hierarchical structure in which the functions in instrument drivers and LabWindows/CVI libraries are grouped.

I

instrument driver	A group of several subprograms related to a specific instrument that reside on disk in a special language-independent format. An instrument driver is used to generate and execute code interactively through menus, dialog boxes, and function panels.

K

KB	Kilobytes of memory.

L

Library Tree	An area in the Workspace window that contains a tree view of the LabWindows/CVI libraries and instruments.

M

MB	Megabytes of memory.
menu	An area accessible from the menu bar that displays a subset of the possible menu items.
menu bar	Mechanism for encapsulating a set of commands. A menu bar appears at the top of the screen and contains a set of menu titles.

O

Output Window Region	An area of the Workspace window in which errors, output, and search result windows appear.

P

panel	Rectangular region of the screen containing a set of controls that accept input from the user and display information to the user. Panels can perform many different functions, such as representing the front panel of an instrument or allowing the user to select a filename.
project	A list of files, usually including a source file, user interface resource file, and header file, that your application uses.
Project Tree	An area of the Workspace window that contains the lists of projects and files in the current workspace.

S

select	To choose the item that the next executed action will affect by moving the input focus (highlight) to a particular item or area.
Source window	A LabWindows/CVI work area in which you edit and execute complete programs. The file extension `.c` designates a file that appears in this window.
Standard Input/Output window	A LabWindows/CVI work area in which output to and input from the screen take place.
standard libraries	The LabWindows/CVI ActiveX, Advanced Analysis (or Analysis), DataSocket, DDE Support, Formatting and I/O, GPIB/GPIB 488.2, RS-232, TCP Support, User Interface, and Utility libraries and the ANSI C Library.
subsidiary DLL	A DLL that an executable or another DLL uses.

U

user interface resource (`.uir`) file	Source code that contains all the objects associated with a user interface. This file includes menu bars, panels, controls, pop-up panels, preferences, images, and fonts. To display user interface objects, an application program must call the User Interface Library to load them from the user interface resource file. A single application program can use multiple user interface resource files.

W

window	A working area that supports operations related to a specific task in the development and execution processes.
Window Confinement Region	An area of the Workspace window that contains open Source, User Interface Editor, and Function Tree Editor windows.
Workspace window	The main work area in LabWindows/CVI; contains the Project Tree, Library Tree, Window Confinement Region, and Output Window Region.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W