LabWindows/CVI

LabWindows/CVI Programmer Reference Manual

NATIONAL INSTRUMENTS™
The Software is the Instrument™

February 1998 Edition
Part Number 320685D-01

Internet Support
E-mail: support@natinst.com
FTP Site: ftp.natinst.com
Web Address: http://www.natinst.com

Bulletin Board Support
BBS United States: 512 794 5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support
512 418 1111

Telephone Support (USA)
Tel: 512 795 8248
Fax: 512 794 5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11,
France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091,
Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, Netherlands 0348 433466, Norway 32 84 84 00,
Singapore 2265886, Spain 91 640 0085, Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200,
United Kingdom 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway Austin, Texas 78730-5039 USA Tel: 512 794 0100

Important Information

Warranty

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

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

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

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

Copyright

Trademarks

CVI™, National Instruments™, the National Instruments logo, `natinst.com`™, and The Software is the Instrument™are trademarks of National Instruments Corporation.

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

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

About This Manual

Chapter 1
LabWindows/CVI Compiler

Chapter 2
Using Loadable Compiled Modules

Chapter 3
Windows 95/NT Compiler/Linker Issues

Chapter 4
Windows 3.1 Compiler/Linker Issues

Chapter 5
UNIX Compiler/Linker Issues

Chapter 6
Building Multiplatform Applications

Chapter 7
Creating and Distributing Standalone Executables and DLLs

Appendix A
Errors and Warnings

Appendix B
Customer Communication

Glossary

Figures

Tables

About This Manual

The *LabWindows/CVI Programmer Reference Manual* contains information to help you develop programs in LabWindows/CVI. The *LabWindows/CVI Programmer Reference Manual* is intended for use by LabWindows users who have already completed the *Getting Started with LabWindows/CVI* tutorial. To use this manual effectively, you should be familiar with *Getting Started with LabWindows/CVI*, the *LabWindows/CVI User Manual*, DOS, Windows, and the C programming language.

Organization of This Manual

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

- Chapter 1, *LabWindows/CVI Compiler*, describes LabWindows/CVI compiler specifics, C language extensions, 32-bit compiler issues, debugging levels, and user protection.
- Chapter 2, *Using Loadable Compiled Modules*, describes the advantages and disadvantages of using compiled code modules in your application. It also describes the kinds of compiled modules available in LabWindows/CVI and includes programming guidelines for modules you generate with external compilers.
- Chapter 3, *Windows 95/NT Compiler/Linker Issues*, describes the different kinds of compiled modules available under LabWindows/CVI for Windows 95/NT and includes programming guidelines for modules you generate with external compilers.
- Chapter 4, *Windows 3.1 Compiler/Linker Issues*, describes the different kinds of compiled modules available under LabWindows/CVI for Windows 3.1 and includes programming guidelines for modules you generate with external compilers.
- Chapter 5, *UNIX Compiler/Linker Issues*, describes the kinds of compiled modules available under LabWindows/CVI for UNIX and includes programming guidelines for modules you generate with external compilers.
- Chapter 6, *Building Multiplatform Applications*, contains guidelines and caveats for writing platform-independent LabWindows/CVI applications. LabWindows/CVI currently runs under Windows 3.1 and Windows 95/NT for the PC, and Solaris 1 and Solaris 2 for the SPARCstation.

- Chapter 7, *Creating and Distributing Standalone Executables and DLLs*, describes how the LabWindows/CVI Run-time Engine, DLLs, externally compiled modules, and other files interact with your executable file. This chapter also describes how to perform error checking in a standalone executable program. You can create executable programs from any project that runs in the LabWindows/CVI environment.
- Chapter 8, *Distributing Libraries and Function Panels*, describes how to distribute libraries, add libraries to a user's **Library** menu, and specify library dependencies.
- Chapter 9, *Checking for Errors in LabWindows/CVI*, describes LabWindows/CVI error checking and how LabWindows/CVI reports errors in LabWindows/CVI libraries and compiled external modules.
- Appendix A, *Errors and Warnings*, contains an alphabetized list of compiler warnings, compiler errors, link errors, DLL loading errors, and external module loading errors generated by LabWindows/CVI.
- Appendix B, *Customer Communication*, contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation.
- The *Glossary* contains an alphabetical list of terms used in this manual and a description of each.
- The *Index* contains an alphabetical list of key terms and topics used in this manual, including the page where each one can be found.

Conventions Used in This Manual

The following conventions are used in this manual.

<> Angle brackets enclose the name of a key on the keyboard—for example, <Shift>.

- A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <Ctrl-Alt-Delete>.

» The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options» Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** options from the last dialog box.

☞	This icon to the left of bold italicized text denotes a note, which alerts you to important information.
⚠	This icon to the left of bold italicized text denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes the names of menus, menu items, parameters, or dialog box buttons.
bold italic	Bold italic text denotes an activity objective, note, caution, or warning.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text from which you supply the appropriate word or value.
`monospace`	Text in this font denotes text or characters that you should literally enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, functions, filenames and extensions, and for statements and comments taken from programs.
`monospace bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen.
`monospace italic`	Italic text in this font denotes that you must enter the appropriate words or values in the place of these items.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, folders, and files.

Related Documentation

You may find the following documentation helpful while programming in LabWindows/CVI:

- *Microsoft Developer Network CD*, Microsoft Corporation, Redmond WA
- *Programmer's Guide to Microsoft Windows 95*, Microsoft Press, Redmond WA, 1995
- Harbison, Samuel P. and Guy L. Steele, Jr., *C: A Reference Manual*, Englewood Cliffs, NJ: Prentice-Hall, Inc., 1995

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help you if you have problems with them. To make it easy for you to contact us, this manual contains comment and technical support forms for you to complete. These forms are in Appendix B, *Customer Communication*, at the end of this manual.

1

LabWindows/CVI Compiler

This chapter describes LabWindows/CVI compiler specifics, C language extensions, 32-bit compiler issues, debugging levels, and user protection.

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, 1991, 1992, 1993 David R. Hanson). It is not an optimizing compiler, but instead focuses on debugging, user protection, and platform independence. Because the compiler is an integral part of the LabWindows/CVI environment and features a limited set of straightforward options, it is also 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	64
Maximum nesting of compound blocks	32
Maximum size of array/struct types	2^{31}

Compiler Options

You can set the LabWindows/CVI compiler options by selecting **Options»Compiler Options** in the Project window. This command opens a dialog box that allows you to set LabWindows/CVI compiler options. For a discussion of these options, refer to the *Compiler Options* section in Chapter 3, *Project Window,* of the *LabWindows/CVI User Manual*.

Compiler Defines

The LabWindows/CVI compiler accepts compiler defines through the **Compiler Defines** command in the **Options** menu of the Project window. For more information, refer to the *Compiler Defines* section in Chapter 3, *Project Window,* of the *LabWindows/CVI User Manual*.

C Language Non-Conformance

LabWindows/CVI for UNIX does not allow you to pass a `struct` as one of a series of unspecified variable arguments. Because of this, `va_arg (ap, type)` is not legal in LabWindows/CVI if `type` is a struct type.

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 the commonly used C extensions in external compilers under Windows 95/NT.

Keywords That Are Not ANSI C Standard

LabWindows/CVI for Windows 3.1 accepts the non-ANSI C keywords `pascal`, `PASCAL`, and `_pascal`, but ignores them.

Calling Conventions (Windows 95/NT Only)

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

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

In Microsoft Visual C/C++, Borland C/C++, and Symantec C/C++, the calling convention normally defaults to `__cdecl` if you do not use a calling convention qualifier. You can,

however, set options to cause the calling convention to default to __stdcall. The behavior is the same in LabWindows/CVI. You can set the default calling convention to either __cdecl or __stdcall using the **Compiler Options** command in the **Options** menu of the Project window. When you create a new project, the default calling convention is __cdecl.

In Watcom C/C++, the default calling convention is not __cdecl or __stdcall. You must use the -4s (80486 Stack-Based Calling) option when you compile a module in Watcom for use in LabWindows/CVI. Refer to the *Compatibility with External Compilers* section in Chapter 3, *Windows 95/NT Compiler/Linker Issues*. The -4s option causes the stack-based calling convention to be the default. In LabWindows/CVI under Watcom compatibility mode, the default calling convention is always the stack-based convention. It cannot be changed. The LabWindows/CVI compiler accepts the __cdecl and __stdcall conventions under Watcom, except that floating point and structure return values do not work in the __cdecl calling convention. National Instruments recommends that you avoid using __cdecl with Watcom.

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

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

National Instruments recommends 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 these qualifiers work in all external compilers. The LabWindows/CVI `cvidef.h` include file defines the following macros, which are designed to work in each external compiler.

```
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 the target type is 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.

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

The `pack pragma` can be used within LabWindows/CVI to specify the maximum alignment factor for elements within a structure. For example, assume the following structure definition:

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

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

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

You can set the maximum alignment as follows:

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

The maximum alignment the compiler applies to a structure is based on the last `pack pragma` statement it sees before the definition of the structure.

Program Entry Points (Windows 95/NT Only)

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

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

C Library Issues

This section discusses special considerations in LabWindows/CVI in the areas of low-level I/O functions and the UNIX C library.

Using the Low-Level I/O Functions

Many functions in the UNIX libraries and the C compiler libraries for the PC are not ANSI C Standard Library functions. In general, LabWindows/CVI implements the ANSI C Standard Library. Under UNIX, you can call UNIX libraries for the non-ANSI C functions in conjunction with LabWindows/CVI.

The low-level I/O functions `open`, `close`, `read`, `write`, `lseek`, and `eof` are not in the ANSI C Standard Library. Under UNIX, these functions are available in the UNIX C library. Refer to Chapter 5, *UNIX Compiler/Linker Issues*, for more information.

Under Windows, you can use these functions along with `sopen` and `fdopen` if you include `lowlvlio.h`.

C Data Types and 32-Bit Compiler Issues

This section introduces the LabWindows/CVI compiler data types and discusses converting 16-bit source code to 32-bit source code.

Data Types

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

Table 1-2. LabWindows/CVI 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}	$2^{31}-1$
unsigned int	32	0	$2^{32}-1$
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 after you complete the steps in Chapter 12, *Converting LabWindows for DOS Applications*, of the *Getting Started with LabWindows/CVI* manual.

In general, if you make 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 and 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 does cause 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 it have the `[b2]` modifier when you pass them as an argument to a Formatting and I/O Library function.

- Uses an `int` variable in a way that requires it 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 it matches a `%d[b2]` or `%i[b2]` specifier, it does not work correctly. Remove the `[b2]` modifier, or declare the variable as `short`.

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

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*** `b`*n* ***modifier.***

All pointers are 32-bit offsets. LabWindows/CVI does not use the `far` pointers that have both a segment selector and an offset, except in 16-bit Windows DLLs under Windows 3.1. LabWindows/CVI for Windows 3.1 calls 16-bit DLLs through a special interface LabWindows/CVI generates from the header file for the DLL. For more information, refer to the *Using 32-Bit Watcom Compiled Modules under Windows 3.1* and *16-Bit Windows DLLs* sections in Chapter 4, *Windows 3.1 Compiler/Linker Issues*.

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. You set the debugging level by selecting **Options»Run Options** in the Project window. Refer to the *Run Options* section in Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual* for information on debugging levels.

User Protection

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

Error categories include pointer protection, dynamic memory protection, library protection, and general protection errors. Each of these categories includes subgroups as described in the following sections.

Array Indexing and Pointer Protection Errors

The pointer protection errors catch invalid operations with pointers and arrays. In this section, these errors 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 sub-expression and an integer sub-expression. LabWindows/CVI generates an error when the pointer sub-expression is

invalid or when the arithmetic operation results in an invalid pointer expression. The following user protection errors involve pointer arithmetic:

- Pointer arithmetic involving uninitialized pointer
- Pointer arithmetic involving 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)
- Pointer arithmetic involving pointer to freed memory
- Pointer arithmetic involving invalid pointer
- Pointer arithmetic involving address of non-array object
- Pointer arithmetic involving pointer to 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 uninitialized pointer value
- Assignment of out-of-bounds pointer expression (assignment of an address before the start, or past the last element, of an array)
- Assignment of pointer to freed memory
- Assignment of invalid pointer expression

Pointer Dereference Errors (Fatal)

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

- Dereference of uninitialized pointer
- Dereference of null pointer
- Dereference of out-of-bounds pointer (dereference using a pointer value before the start, or past the end, of an array)
- Dereference of pointer to freed memory
- Dereference of invalid pointer expression
- Dereference of data pointer for use as a function
- Dereference of function pointer for use as data

- Dereference of a pointer to an *n*-byte type where less than *n* bytes exist in the object
- Dereference of unaligned pointer (UNIX only)

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 uninitialized pointer
- Comparison involving null pointer
- Comparison involving invalid pointer
- Comparison of pointers to different objects
- Pointer comparison involving address of 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 uninitialized pointer
- Subtraction involving null pointer
- Subtraction involving invalid pointer
- Subtraction of pointers to different objects
- Pointer subtraction involving address of 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 occurs only when casting 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 uninitialized pointer
- Attempt to free pointer to freed memory
- Attempt to free invalid pointer expression
- Attempt to free 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:

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

The missing return value error means that a non-void function (one you do not declare with `void` return type) returned, but did not returned a value.

Library Protection Errors

Library functions sometimes 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:

- Null pointer argument to library function
- Uninitialized pointer argument to library function
- Passing a pointer to freed memory to a library function
- Array argument too small
- Passing by reference a scalar argument to a library function that expects an array

- Missing terminating null in string argument
- 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 Break on Library Errors option in the **Run Options** command in the **Options** menu of the Project 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 Break on Library Errors feature.

Disabling Library Errors at Run-Time

The Break on Library Errors option in the **Run Options** command in the **Options** menu of the Project 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. Use of 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 could be useful in the following situation: LabWindows/CVI reports erroneous run-time errors because you set a pointer to dynamic memory in a source module and you then resize it in an object module. The following steps describe how this error occurs:

1. You declare a pointer in a source module 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 the debugging level is **None**. 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 occurrence of the pragma.

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

These two pragmas enable and disable run-time checking for a particular library function throughout the module in which they appear. You can use them 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 function*** `SetBreakOnLibraryErrors`***. To disable the run-time checking of argument expressions for a particular call to a library function, use the Utility Library function*** `SetBreakOnProtectionErrors`**.**

☞ **Note** ***You cannot use pragmas to disable protection for the functions in the statically linked libraries including User Interface, RS-232, TCP, DDE, Formatting and I/O, Utility, X Property, 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 parenthesis.

C programmers occasionally have 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 one, 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 the memory can be used. 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* discussion in this 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 its corresponding dynamic memory.

A case exists where the garbage collection fails to retrieve all the memory it allocated. This occurs when you pass the return value of one function to another function, the return value is a pointer to dynamically allocated memory, and you do not assign the pointer to a variable in the argument expression. The following is an example:

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

This call passes the return value from `malloc` to `MyFunc` but does not assign it 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. Some examples follow:

```
{
   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 for passing function parameters and storing automatic local variables. You can set the maximum stack size by selecting the **Options»Run Options** in the Project window. Table 1-3 shows the stack size ranges LabWindows/CVI supports.

Table 1-3. Stack Size Ranges for LabWindows/CVI

Platform	Minimum	Default	Maximum
Windows 3.1	4 KB	40 KB	40 KB
Windows 95/NT	100 KB	250 KB	1 MB
Solaris 1 for Sun	100 KB	250 KB	5 MB
Solaris 2 for Sun	100 KB	250 KB	5 MB

☞ **Note** ***For LabWindows/CVI for Windows 3.1, the actual stack size approaches 64 KB when you set the Debugging level to None.***

Include Paths

The **Include Paths** command in the **Options** menu of the Project window specifies the directory search path for include files. The Include Paths dialog box has two lists, one for include paths specific to the project, and one for paths not specific to 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. LabWindows/CVI also searches that directory for include files.

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 `cvi\toolslib` directory
6. The `cvi\instr` directory
7. The `cvi\include` directory
8. The `cvi\include\ansi` directory
9. The VXI*plug&play* include directory
10. The `cvi\sdk\include` directory (Windows 95/NT only)

2

Using Loadable Compiled Modules

This chapter describes the advantages and disadvantages of using compiled code modules in your application. It also describes the kinds of compiled modules available in LabWindows/CVI and includes programming guidelines for modules you generate with external compilers.

Refer to Chapter 3, *Windows 95/NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, or Chapter 5, *UNIX Compiler/Linker Issues*, in this manual for more information on platform-specific programming guidelines for modules that external compilers generate.

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 on the PC, or a `.o` file under UNIX, that contains one object module
- A `.lib` file on the PC, or a `.a` file under UNIX, that contains one or more object modules
- A `.dll` file that contains a Windows DLL (Windows only)

You can create any of these compiled modules in LabWindows/CVI under Windows 95/NT, or using a compatible external compiler. Under Windows 3.1, LabWindows/CVI can create only `.obj` files. Under UNIX, LabWindows/CVI can create only `.o` files.

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

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

- Compiled modules run faster than source modules. Compiled modules do not contain the debugging and user protection code LabWindows/CVI generates when it compiles source modules. Compiled modules you generate in external compilers can run faster because of optimization.
- LabWindows/CVI recompiles the source modules in a project each time you open the project. Also, if an instrument driver program file is a source module, LabWindows/CVI recompiles it each time you load the instrument driver. LabWindows/CVI does not recompile compiled modules when you open a project or load an instrument driver.
- In standalone executables, you can dynamically load compiled modules but not source modules.
- You can install compiled modules, but not source modules, into the **Library** menu.
- 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. It 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 it also can be a software utility. The *Using Instrument Drivers* and *Instrument Menu* sections of Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual* describe how to use instrument drivers.

To develop and debug an instrument driver, load its program file into LabWindows/CVI as a source file. After you finish debugging it, you can compile the program file into an object file or a Windows 95/NT DLL. The next time you load the instrument driver, LabWindows/CVI loads the compiled module, which loads and runs faster that the source module.

Refer to the *LabWindows/CVI Instrument Driver Developers Guide* for information on how to create an instrument driver.

If the instrument driver program file is a compiled module, it must adhere to the requirements outlined for each operating system in Chapter 3, *Windows 95/NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, and Chapter 5, *UNIX Compiler/Linker Issues*, of this manual.

Using a Loadable Compiled Module as a User Library

You can install your own libraries into the **Library** menu. 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. Refer to the *Using Instrument Drivers* and the *Instrument Menu* sections of Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual* for more information. 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 **Unload** command in the **Instrument** menu, but you cannot unload user libraries. You cannot edit and recompile user libraries while they are loaded.

Install user libraries by selecting the **Library Options** command in the **Project Options** menu. The next time you run LabWindows/CVI, the libraries load automatically and appear at the bottom of the **Library** menu.

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 **Library Options** 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** menu.

User libraries must adhere to the requirements outlined for the target operating system. Chapter 3, *Windows 95/NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, and Chapter 5, *UNIX Compiler/Linker Issues*, of this manual, discuss operating system requirements.

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 under Windows 95/NT, you must include the DLL import library (`.lib`) file in the project list rather than the DLL.***

Even when you include a source module in the project list, you can instruct LabWindows/CVI to create an object module on disk when it compiles the file instead of debuggable code in memory. To do this, double click in the O column next to the source file in the Project window.

Compiled modules must adhere to the requirements outlined for the target operating system. Chapter 3, *Windows 95/NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, and Chapter 5, *UNIX Compiler/Linker Issues*, of this manual, discuss operating system requirements.

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`. Refer to Chapter 8, *Utility Library,* of the *LabWindows/CVI Standard Libraries Reference Manual* for more information on using these functions.

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 95/NT DLL. External modules must adhere to the requirements outlined for the target operating system. Chapter 3, *Windows 95/NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, and Chapter 5, *UNIX Compiler/Linker Issues*, of this manual, discuss operating system requirements.

Notification of Changes in Run State

You might have to notify certain compiled modules whenever your program starts, suspends, continues, or stops. For example, if a compiled module has asynchronous callbacks, you must prevent the callbacks from executing when program execution suspends at a breakpoint. LabWindows/CVI has a callback mechanism you can use to inform a compiled module of changes in the program status.

To notify a compiled module of changes in the run state, add a function with the name `__RunStateChangeCallback` to the compiled module. LabWindows/CVI automatically installs the callback for you.

The run state change callback must be in a compiled file, not in a source file. More than one compiled module can contain functions with this name, because LabWindows/CVI never enters it into the global name space. The prototype for the callback is as follows:

```
void CVICALLBACK __RunStateChangeCallback(int action)
```

`libsupp.h` defines the actions in the following enumerated type:

```
enum {
      kRunState_Start,
      kRunState_Suspend,
      kRunState_Resume,
      kRunState_AbortingExecution,
      kRunState_Stop,
```

```
      kRunState_EnableCallbacks,
      kRunState_DisableCallbacks
};
```

The following examples show typical program state changes.

Example 1

```
kRunState_Start
kRunState_EnableCallbacks
   /* user program execution begins */
.
.
.
   /* a breakpoint or run-time error occurs, or user presses the
      Terminate Execution key combination */
kRunState_DisableCallbacks
kRunState_Suspend
   /* program execution suspends; CVI environment resumes */
.
.
.
   /* user requests the execution be resumed, through the "Continue",
      "Step Over", etc., commands */
kRunState_Resume
kRunState_EnableCallbacks
   /* user program execution resumes */
   .
   .
   .
   /* user program execution completes normally */
kRunState_DisableCallbacks
kRunState_Stop
```

Example 2

```
kRunState_Start
kRunState_EnableCallbacks
   /* user program execution begins */
   .
   .
   .
   /* a breakpoint or run-time error occurs, or user presses the
      Terminate Execution key combination */
kRunState_DisableCallbacks
kRunState_Suspend
   /* program execution suspends; CVI environment resumes */
   .
   .
   .
   /* user selects the Terminate Execution command */
kRunState_DisableCallbacks /* even though callbacks already
                              disabled */
kRunState_AbortingExecution
   /* long jump out of user program */
kRunState_DisableCallbacks /* even though callbacks already
                              disabled */
kRunState_Stop
```

Note ***A*** `Resume` ***notification does not always follow a*** `Suspend` ***notification. A*** `Stop` ***notification can follow a*** `Suspend` ***notification without an intervening*** `Resume` ***notification.***

Note ***Run state change callbacks do not work if you link your program in an external compiler. Also, external compilers report link errors if you have multiple run state change callbacks.***

Using Run State Change Callbacks in a DLL

You can include one or more run state change callbacks in a DLL. To do so, you must build the DLL in the LabWindows/CVI development environment, and each run state change callback must be in a separate object or static library file in the DLL project. If you include a run state change callback in a DLL, or in an object or static library file that another user might include in a DLL, take special care in two areas:

- Use caution when you call into other DLLs in response to a `kRunState_Stop` message. When you use your DLL in a standalone executable, the DLL receives the `kRunState_Stop` message when the executable terminates. The order in which

Windows 95/NT unloads DLLs at process termination is not well-defined. Therefore, the DLL you call into might no longer be loaded. This can cause a general protection fault.

Nevertheless, when you use your DLL in a program in the LabWindows/CVI development environment, it is often necessary to call into DLLs to release resources after each run. To solve this dilemma, use conditional code to release resources only if you are running in the LabWindows/CVI development environment. An example follows.

```
#include <utility.h>

switch (runState)
   {
   case kRunState_Stop:
      if (! InStandaloneExecutable())
         { /* call into other DLLs to release resources */ }
      /* release resources, including unloading DLLs */
      break;
   }
```

It is always safe to call into the LabWindows/CVI Run-time Engine in a run state change callback.

- If your DLL uses global variables that can become stale after each program execution in the LabWindows/CVI development environment, re-initialize the variables in response to the `kRunState_Start` or `kRunState_Stop` message. For example, memory that you allocate using LabWindows/CVI ANSI C functions such as `malloc` or `calloc` is no longer valid when you restart your program. If your DLL has global variables that point to allocated memory, set those pointers to `NULL` in response to the `kRunState_Start` or `kRunState_Stop` message.

Compiled Modules that Contain Asynchronous Callbacks

A compiled module can call a source code function asynchronously. This can happen through interrupts or signals. In Windows 95/NT, the compiled module can call the source code function from a thread other than the main thread. The call takes place asynchronously with respect to the normal execution of the source code in your program.

The execution and debugging system in the LabWindows/CVI development environment is not prepared to handle this asynchronous execution. Consequently, the compiled module must announce to LabWindows/CVI that it is calling asynchronously into source code. It does this by calling `EnterAsyncCallback` before calling the function, and calling `ExitAsyncCallback` after calling the function. `EnterAsyncCallback` and `ExitAsyncCallback` have one parameter, which is a pointer to a buffer of size `ASYNC_CALLBACK_ENV_SIZE`. You must pass the same buffer into `ExitAsyncCallback` that you passed into `EnterAsyncCallback` because the buffer stores state information. The definition of `ASYNC_CALLBACK_ENV_SIZE` and the prototypes for these two functions are in `libsupp.h`.

3

Windows 95/NT Compiler/Linker Issues

This chapter describes the different kinds of compiled modules available under LabWindows/CVI for Windows 95/NT and includes programming guidelines for modules you generate with external compilers.

Under Windows 95/NT, the LabWindows/CVI compiler is compatible with four external 32-bit compilers: Microsoft Visual C/C++, Borland C/C++, Watcom C/C++, and Symantec C/C++. This manual refers to the four compilers as the *compatible external compilers*.

In LabWindows/CVI under Windows 95/NT, you can do the following:

- Load 32-bit DLLs, through 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 any of the four 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 any of the four compatible external compilers
- Call Windows Software Development Kit (SDK) functions

This chapter discusses these capabilities.

Loading 32-Bit DLLs under Windows 95/NT

Under Windows 95/NT, 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

Under Windows 95/NT, 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.

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.

Link Errors when Using DLL Import Libraries

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

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 95/NT.

16-Bit DLLs Not Supported

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

Run State Change Callbacks in DLLs

You can include run state change callbacks in DLLs you build in LabWindows/CVI. When running a program in LabWindows/CVI, a run state change callback receives notification when the program starts, suspends, resumes, and stops. If you include a run state change

callback in a DLL, you must take special care. Refer to the *Notification of Changes in Run State* section in Chapter 2, *Using Loadable Compiled Modules*, of this manual, for a detailed discussion of run state change callbacks.

DllMain

Each DLL can have a `DllMain` function, except that the Borland compiler uses `DllEntryPoint` as the name. The operating system calls the `DllMain` function with various messages. To generate the template for a `DllMain` function, use the **Insert Constructs** command in the **Edit** menu of 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 95/NT 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 run state change callback, as long as you do so before calling `CloseCVIRTE`.

Releasing Resources when a DLL Unloads

When a program terminates, the operating system disposes resources your DLL allocates. If your DLL remains loaded throughout program execution, it does not need to dispose resources explicitly when the system unloads it 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. It can release resources in the `DllMain` function in response to the `PROCESS_DETACH` message. The DLL can also 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 previous 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 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, and DDE Libraries.***

Generating an Import Library

If you do not have a DLL import library or if the one you have contains references the DLL does not export, 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 convention 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 the **Generate DLL Import Library** command in the **Options** menu.

Default Unloading/Reloading Policy

Some fundamental differences exist in the way Windows 95/NT and Windows 3.1 handle a DLL that multiple processes use.

Windows 95/NT creates a separate data space for each process that uses the DLL. Windows 3.1 creates only one data space for all processes that use the DLL.

Windows 95/NT notifies a DLL each time a process loads or unloads it. Windows 3.1 does not notify a DLL each time a process loads or unloads it. Windows 3.1 notifies the DLL only when the first process loads it and the last process unloads it.

LabWindows/CVI for Windows 95/NT unloads DLLs, by default, after each execution of a user program in the development environment. This behavior more accurately simulates what happens when you execute a standalone executable, and it is more suitable for Windows 95/NT DLLs that rely on load/unload notification on each execution of a program. You can change the default behavior by turning off the Unload DLLs After Each Run option in the Run Options dialog box of the Project window. National Instruments recommends, however, that you leave the default behavior in effect.

Compatibility with External Compilers

LabWindows/CVI for Windows 95/NT can be compatible at the object code level with any of the four compatible external compilers (Microsoft Visual C/C++, Borland C/C++, Watcom C/C++, and Symantec C/C++). Because these compilers are not compatible 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 Your Compatible Compiler

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

You can see which compatible compiler is active in LabWindows/CVI by selecting the **Compiler Options** command in the **Options** menu of the Project window.

Object Files, Library Files, and DLL Import Libraries

If you create an object file, library file, or DLL import library in LabWindows/CVI, you can use the file only in the current compatible compiler or in a copy of LabWindows/CVI that you installed with the same compatibility choice. For detailed information on using LabWindows/CVI-generated object and static library files in external compilers, refer to the *Using LabWindows/CVI Libraries in External Compilers* section later in this chapter.

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 or in a copy of LabWindows/CVI that you installed with the same compatibility choice. 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 the **Generate DLL Import Library** command in the **Options** menu.

Make sure the calling conventions of the function declarations in the include file match the calling convention 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 following are areas in which the DLLs that external compilers create are not fully compatible.

Structure Packing

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

If your DLL API uses structures, you 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;

typdef 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. The other compilers always use 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. Example:

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

Returning Floats and Doubles

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

Enum Sizes

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

Long Doubles

In Borland C/C++, `long double` values are 10 bytes. In the other compilers, 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 the 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 are the most important differences you might encounter:

- `wchart_t` is only one-byte in LabWindows/CVI.
- 64-bit integers do not exist in LabWindows/CVI.
- `long double` values are 10 bytes in Borland C/C++ but 8 bytes in LabWindows/CVI.
- You cannot use structured exception handling in LabWindows/CVI.
- You cannot use the Watcom C/C++ `__cdecl` calling convention in LabWindows/CVI for functions that return `float` or `double` scalar values or structures. In Watcom, `__cdecl` is *not* the default calling convention.
- LabWindows/CVI does not define `_MSC_VER`, `__BORLANDC__`, `__WATCOMC__`, and `__SC__`. The external compilers each define one of these macros. If you port 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 for Windows 95/NT:

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

Required Preprocessor Definitions

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

```
_NI_mswin32_
```

Multithreading and the LabWindows/CVI Libraries

Although the LabWindows/CVI environment is not multithreaded, you can use LabWindows/CVI Libraries in the following multithreaded contexts:

- When you call the LabWindows/CVI Libraries from a multithreaded executable you create in LabWindows/CVI or in an external compiler.
- When you call the LabWindows/CVI Libraries from a DLL that a multithreaded executable loads. You can create the DLL in LabWindows/CVI or in an external compiler.
- When you call the LabWindows/CVI Libraries from an object or static library file that you dynamically load in a multithreaded executable. You can create the object or library file in LabWindows/CVI or in an external compiler.

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

For detailed information on how to use the LabWindows/CVI User Interface Library in a multithreaded program, refer to Chapter 3, *Programming with the User Interface Library*, in the *LabWindows/CVI User Interface Reference Manual*.

Using LabWindows/CVI Libraries in External Compilers

Under Windows 95/NT, you can use the LabWindows/CVI libraries in any of the four compatible external compilers. You can create executables and DLLs that call the LabWindows/CVI libraries. 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 `cvi\extlib` directory contains DLL import libraries and a startup library, all compatible with your external compiler. Never use the `.lib` files in the `cvi\bin` directory in an external compiler.

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

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

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

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

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

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

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

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

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 which */
                 /* calls main()                        */
```

Include Files for the ANSI C Library and the 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 `cvi\include` directory, the LabWindows/CVI ANSI C include files are in the `cvi\include\ansi` directory. Thus, you can specify `cvi\include` as an include path in your external compiler 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, refer to the *Setting Up Include Paths for LabWindows/CVI, ANSI C, and SDK Libraries* section later in this chapter.

Standard Input/Output Window

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

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

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. This is true whether you run your program in the LabWindows/CVI development environment or as a standalone executable.

When you link your program in an external compiler, the external compiler does not make such a table available to the User Interface Library. To resolve callback references, you must use LabWindows/CVI to generate an object 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 the **External Compiler Support** command in the **Build** menu of the Project window. A dialog box appears.
3. In the UIR Callbacks Object File control, enter the pathname of the object file you want to generate. When you click on the **Create** button, LabWindows/CVI generates the object 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 object file to reflect the changes.
4. Include this object 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. Refer to the *Calling InitCVIRTE and CloseCVIRTE* section later in this chapter.

Linking to Callback Functions Not Exported from a DLL

Normally, the User Interface Library searches for callback functions only in the table of functions in the executable. When you load a panel or menu bar from a DLL, you might want to link to non-static callback functions 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. Refer to Chapter 4, *User Interface Library Function Reference*, of the *LabWindows/CVI User Interface Reference Manual* for detailed information on `LoadPanelEx` and `LoadMenuBarEx`.

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 an object file that contains the necessary table as follows.

1. Create a LabWindows/CVI project that contains the `.uir` files your DLL loads, if you do not already have one.
2. Select the **External Compiler Support** command in the **Build** menu of the Project window. A dialog box appears.
3. In the UIR Callbacks Object File control, enter the pathname of the object file you want to generate. When you click on the **Create** button, LabWindows/CVI generates the object 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 object file to reflect the changes.

4. Include this object file in the external compiler project you use to create the DLL.
5. You must call `InitCVIRTE` and `CloseCVIRTE` in your `DLLMain` function. Refer to the *Calling InitCVIRTE and CloseCVIRTE* section later in this chapter.

Resolving References from Modules Loaded at Run-Time

Note ***This section does not apply unless you use*** `LoadExternalModule` ***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. This is true whether you run your program in the LabWindows/CVI development environment or as a standalone executable.

When you link your program in an external compiler, the external compiler does not make such a table available to the Utility Library. LabWindows/CVI provides ways to help you create the symbol table easily.

Resolving References to the LabWindows/CVI Run-Time Engine

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

- Include `cvi\extlib\refsym.obj` 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, TCP, or Utility Library.
- Include `cvi\extlib\arefsym.obj` in your external compiler project if your run-time modules refer to any symbols in the ANSI C library. If you have to use this object file and you use Borland C/C++ to create your executable, you must choose Static Linking for the Standard Libraries. In the Borland C/C++ IDE, you can do this in the New Target and Target Expert dialog boxes.

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. 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 `cvi\sdk\include\basicsdk.h` file.

Execute the **External Compiler Support** command in the **Build** menu of the Project window. A dialog box appears. Enable the Using Load External Module option. Enable the Other Symbols checkbox if it is not already enabled. Enter the pathname of the include file in the Header File control. Enter the pathname of the object file to generate in the Object File control. Click on the **Create** button to the right of the Object File control.

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

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. Refer to Chapter 8, *Utility Library*, of the *LabWindows/CVI Standard Libraries Reference Manual* for detailed information on `LoadExternalModuleEx`.

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 can do this using the technique that the previous section, *Resolving References to Symbols Not in Run-Time Engine*, describes. You must call `InitCVIRTE` and `CloseCVIRTE` in your `DLLMain` function. Refer to the *Calling InitCVIRTE and CloseCVIRTE* section later in this chapter.

Run State Change Callbacks Are Not Available in External Compilers

When you use a compiled module in LabWindows/CVI, you can arrange for LabWindows/CVI to notify the module of a change in the execution status such as start, stop, suspend, or resume. You do this through a callback function that is always named `__RunStateChangeCallback`. The *Notification of Changes in Run State* section, in Chapter 2, *Using Loadable Compiled Modules*, of this manual, describes this in detail.

The run state change callback capability in LabWindows/CVI is necessary because the LabWindows/CVI development environment executes your program as part of the LabWindows/CVI process. When your program terminates, the operating system does not release resources as it does when a process terminates. LabWindows/CVI attempts to release

resources your program allocated, but your compiled module might have to do more. Also, if the program suspends for debugging purposes, your compiled module might have to disable interrupts.

When you run an executable created in an external compiler, it always executes as a separate process, even when you debug it. Thus, the run state change callback facility is not necessary and does not work. External compilers report link errors when you define `__RunStateChangeCallback` in more than one object file. If you include a run state change callback in a compiled module that you intend to use both in LabWindows/CVI and an external compiler, it is a good idea to put the callback function in a separate source file and create a `.lib` file instead of a `.obj` file.

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 (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 pvReserved)
{
   if (fdwReason == DLL_PROCESS_ATTACH)
      {
      if (InitCVIRTE (hinstDLL, 0, 0) == 0)
          return 0; /* out of memory */
      /* your other ATTACH code  */
      }
    else if (fdwReason == DLL_PROCESS_DETACH)
      {
      /* your other DETACH code */
      CloseCVIRTE ();
      }
      return 1;
}
```

☞ **Note** ***It is harmless, but unnecessary, to call these functions when you link your executable in LabWindows/CVI for Windows 95/NT.***

Watcom Stack Based Calling Convention

When you use the LabWindows/CVI libraries in the Watcom compiler, you must set the default calling convention to the 80486 Stack Based calling convention. In the command line compiler, this is the `-4s` option. In the Watcom IDE, you can set the default calling convention by using the **Options»C Compiler Switches** command. The option is in the Target Processor section of the Memory Model and Processor Switches section of the dialog box. If you do not set this option, the Watcom linker reports undefined references to the LabWindows/CVI run-time libraries.

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 several 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 each compiler.

Microsoft Visual C/C++

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 using the **Input** category in the **Link** tab of the Project Settings dialog box.

Borland C/C++

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

Watcom C/C++

Like Visual C, at least one object file must contain a default library directive to cause the C library to be linked in. In addition, Watcom also requires a default library directive for floating-point support.

If you do not include in your project any object files with the required directives, add the following libraries, in the order shown, to the Libraries setting in the Windows Linking Switches dialog box:

```
clib3s
math387
noemu387
```

Symantec C/C++

Each object file must have the default library directive for the C library. You must explicitly add the Symantec C library to your project. The library filename is `snn.lib` and it is in the `lib` subdirectory under the Symantec installation directory.

Borland Static versus Dynamic C Libraries

When you link a Borland C/C++ 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 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.

Borland C++ Builder

You cannot use your LabWindows/CVI object or static library files in the Borland C++ Builder.

Watcom Pull-in References

The Watcom linker does not automatically link the startup code into your application or DLL. Instead, it requires the module that contains `main`, `WinMain`, or `DllMain` to reference a special symbol that the appropriate startup code module resolves. The Watcom compiler automatically generates a reference to the special symbol into any module that contains `main`, `WinMain`, or `DllMain`. This symbol is `__DLLstart_`, `_wstart2_`, or `_cstart_`, depending on whether the project is for a DLL, Windows application, or console application, respectively. Object modules compiled in LabWindows/CVI do not contain such references. LabWindows/CVI cannot generate the correct reference because it makes no distinction between console and non-console applications.

You must include the symbol reference in your object file explicitly. For example, if your module contains the `main` function, you can generate the correct symbol reference by adding the following to the source code for the module:

```
extern int _cstart_;
void *dummy = &_cstart_;
```

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 `cvi\include` and `cvi\sdk\include` directories. Ensure that these directories have priority over the default paths for the compiler's C library and SDK library include files.

You must choose the compiler options carefully. LabWindows/CVI tries to work with the default options for each compiler as much as possible. In some cases, however, you have to choose options that override the defaults. In other cases you must accept the defaults.

Microsoft Visual C/C++

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++

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)

Watcom C/C++

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

`-ei` (Force Enums to Type Int)

`-bt=nt` (Target Platform is Windows 95/NT)

`-mf` (Flat Memory Model)

`-4s` (80486 Stack-Based Calling)

`-s` (Disable Stack Depth Checking)

`-j` (Change Char Default to Signed)

`-fpi87` (Generate In-Line 80x87 Code)

If your external object calls `LoadExternalModule` or `LoadExternalModuleEx`, you must also add the following compiler option:

`-d__NO_MATH_OPS`

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

`-Zp` (Structure Alignment)

Symantec C/C++

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

`-mn` (Windows 95/NT Memory Model)

`-f` (Generate In-Line 80x87 Code)

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

`-a` (Struct Alignment)

`-P` (Use Pascal Calling Convention)

`-s` (Check Stack Overflow)

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

Creating Executables in LabWindows/CVI

You can create true 32-bit Windows executables in LabWindows/CVI for Windows 95/NT. In LabWindows/CVI for Windows 3.1, you run standalone programs using a special executable file that contains the LabWindows/CVI run-time libraries. If you run more than one program at a time, Windows 3.1 loads extra copies of this special executable into memory. Under Windows 95/NT, the LabWindows/CVI run-time libraries come in DLL form. Standalone 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 95/NT loads only one copy of the DLL.

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

Creating DLLs in LabWindows/CVI

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

You must have a separate project for each DLL you want to create. Select **Dynamic Link Library** from the submenu attached to the **Target** command in the **Build** menu of the Project window. When you select **Dynamic Link Library**, the **Create Dynamic Link Library** command appears below the **Target** command in the **Build** menu. Refer to Chapter 3, *Project Window*, in the *LabWindows/CVI User Manual*, for detailed information on the **Create Dynamic Link Library** command.

You can debug the DLLs you create in LabWindows/CVI. Refer to the *DLL Debugging (Windows 95/NT Only)* section in Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual*, for more information.

Customizing an Import Library

If you have 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 from the DLL, not variables.

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 execute the **Generate DLL Import Source** command in the **Options** menu.

After you have generated 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 **Static Library** from the submenu attached to the **Target** command in the **Build** menu of the Project window. Execute the **Create Static Library** command.

☞ **Note** ***This 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 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
- Marking 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 chapter also discusses these differences.

Some of the information in this section is very technical and complex. Recommendations on the best approaches to these issues are at the end of the section. 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` or Watcom stack-based calling convention to declare the functions you want to export. If, however, 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. This is true whether or not you choose to make `__stdcall` the default calling convention for your

project. You must use the __stdcall keyword in the declarations in the include file you distribute with the DLL.

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 following are examples of using the DLLSTDCALL macro.

```
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 two ways: the include file method and the qualifier method.

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 Create Dynamic Link Library dialog box, you 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.

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 also recognize __export. LabWindows/CVI recognizes both. The cvidef.h include file defines the DLLEXPORT

macro to resolve differences among compilers and platforms. The following are examples of using the DLLEXPORT macro:

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

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

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

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

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

☞ **Note** ***If you include in your DLL project 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.***

Marking Imported Symbols in Include File Distributed with 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 that 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 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 several different ways:

- You can avoid exporting variables from DLLs, and thereby eliminate the need to use import qualifiers. For each variable you want to export, you can create functions to get and set its value or a function to return a pointer to the variable. You do not have to use import qualifiers for functions. This is the simplest approach and works in LabWindows/CVI. However, it 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 by using the **Compiler Defines** command in the **Options** menu of the Project window.

Recommendations

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

- Use the `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 its value or a function to return a pointer to the variable. Do not use import qualifiers in the include file.

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

- You can distribute with your DLL the same include file that you include in the source files 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 any of the four compatible external compilers.
- You can use your DLL in Microsoft Visual Basic or other non-C environments.

Automatic Inclusion of Type Library Resource for Visual Basic

The **Create Dynamic Link Library** command gives you the option to automatically create a Type Library resource and include it in the DLL. When you use this option, Visual Basic users can call the DLL without having to use a header file 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 Windows help file from it using the **Generate Windows Help** command in the **Options** menu of the Function Tree Editor window. The **Create Dynamic Link Library** command provides an option to include links into the Window 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 Dynamic Link Library** command imposes 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 for Windows 95/NT. Static libraries are libraries in the traditional sense—a collection of object files—as opposed to a dynamic link library or an import library. You can use just one project to create static library files that work with all four compatible external compilers, but only if you include no object or library files in the project.

You must have a separate project for each static library you want to create. Select **Static Library** from the submenu attached to the **Target** command in the **Build** menu of the Project window. When you select the **Static Library** option, the **Create Static Library** command appears below the **Target** command in the **Build** menu. Refer to Chapter 4, *Source, Interactive Execution and Standard Input/Output Windows*, of the *LabWindows/CVI User Manual* for detailed information on the **Create Static Library** command.

☞ **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 all four compatible external compilers.***

Creating Object Files in LabWindows/CVI

You can create an object file in LabWindows/CVI in one of two ways:

- Include a source (`.c`) file in your project. Enable the Compile into Object option for the source file by double-clicking in the space next to the filename in the Project window under the column marked “O”. Compile the file.
- Open a source (`.c`) file and select the **Create Object File** command in the **Options** menu of the Source window.

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

☞ **Note** ***Do not set the default calling convention to*** `__stdcall` ***if you want to create a static object for all four compatible external compilers.***

Calling Windows SDK Functions in LabWindows/CVI

You can call Windows SDK Functions in LabWindows/CVI for Windows 95/NT. If you install the LabWindows/CVI full development system from CD-ROM, you can 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 the **Windows SDK** command in the **Help** menu of any LabWindows/CVI window.

Windows SDK Include Files

You must include the SDK include files *before* the LabWindows/CVI include files. In 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 `cvi\sdk\include` directory. The LabWindows/CVI compiler automatically searches the `cvi\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. For more information, refer to the *Setting Up Include Paths for LabWindows/CVI, ANSI C, and SDK Libraries* section later in this chapter.

The number of SDK include files is very large. Normally, you have to 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 by using the **Compiler Defines** command in the **Options** menu of the Project window.

Using Windows SDK Functions for User Interface Capabilities

The LabWindows/CVI User Interface Library uses the Windows SDK. It is not designed to be used in programs that attempt to build other user interface objects at the SDK level. While no specific restrictions exist 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.

Using Windows SDK Functions to Create Multiple Threads

Although you can use the Windows SDK Functions to create multiple threads in a LabWindows/CVI program, the LabWindows/CVI development environment cannot handle multiple threads. For instance, if your main program terminates without destroying the threads, they do not terminate. Also, the LabWindows/CVI libraries are not multithread safe when you run a program in the LabWindows/CVI development environment.

For information on using the LabWindows/CVI libraries in a multithreaded executable, refer to the *Multithreading and the LabWindows/CVI Libraries* section earlier in this chapter.

Automatic Loading of SDK Import Libraries

All the SDK functions are in DLLs. LabWindows/CVI and the four 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 three import libraries:

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

LabWindows/CVI for Windows 95/NT automatically loads these three 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 `cvi\sdk\sdkfuncs.txt` file for associations of SDK import libraries to SDK functions. The import libraries are in the `cvi\sdk\lib` directory.

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

The rules for using SDK include files are not the same as the rules for using ANSI C standard library include files, which in turn are different than the rules for using the LabWindows/CVI library include files. Refer to the *Include Files for the ANSI C Library and the LabWindows/CVI Libraries* and *Windows SDK Include Files* sections earlier in this chapter.

You must set up your include paths differently depending on the environment in which you compile and link. A discussion of each case follows.

Compiling in LabWindows/CVI for Linking in LabWindows/CVI

Use the LabWindows/CVI SDK and ANSI C include files. You do not have 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. Using the **Include Paths** command in the **Options** menu of the Project window, add the following as explicit include paths at the beginning of the project-specific list:

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

Compiling in an External Compiler for Linking in an External Compiler

Use the SDK and ANSI C include files from the external compiler. This happens automatically. Specify the following directories as include paths in the external compiler for the LabWindows/CVI library include files.

```
cvi\include
```

Compiling in an External Compiler for Linking in LabWindows/CVI

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

```
cvi\include
cvi\include\ansi
cvi\sdk\include
```

Handling Hardware Interrupts under Windows 95/NT

Under Windows 3.1, you can handle hardware interrupts in a DLL. Under Windows 95, you must handle hardware interrupts in a VxD. Under Windows NT, 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 Device Driver Developer Kit (DDK).

Under Windows 3.1, it is extremely difficult to call source code into LabWindows/CVI at interrupt time. Making such a call is easier under Windows 95/NT. Under Windows 95/NT, 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. It 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 95 and another for Windows NT.

The files are in `cvi\vxd\win95` and `cvi\vxd\winnt`. The file `template.doc` in each directory contains some basic information.

4

Windows 3.1 Compiler/Linker Issues

This chapter describes the different kinds of compiled modules available under LabWindows/CVI for Windows 3.1 and includes programming guidelines for modules you generate with external compilers.

Using Modules Compiled by LabWindows/CVI

You can generate a compiled `.obj` or `.o` module from a source file within LabWindows/CVI using the **Create Object File** command in the **Options** menu of a Source window. You can then use the compiled module in any of the methods described in the *About Loadable Compiled Modules* section in Chapter 2, *Using Loadable Compiled Modules*, of this manual.

Using 32-Bit Watcom Compiled Modules under Windows 3.1

You must adhere to the following rules for a 32-bit Watcom compiled module (`.obj` or `.lib` file):

- You can call LabWindows/CVI library functions.
- If you make a call to the ANSI C Standard Library, you must include the LabWindows/CVI header files instead of the Watcom header files.
- You cannot call Watcom C library functions outside the scope of the ANSI C Standard Library.
- You can call `open`, `close`, `read`, `write`, `lseek`, or `eof`, but you must include `lowlvlio.h` from LabWindows/CVI.
- You cannot call functions in the Windows Software Development Kit (SDK), install interrupts, perform DMA, or access hardware directly. These tasks must be done with a Dynamic Link Library (DLL). The exception to this is that you can use the `inp` and `outp` functions.
- You cannot define a function as `PASCAL`, `pascal`, or `_pascal` if you intend to call it from source code in LabWindows/CVI. Also, you cannot use any non-ANSI-C-standard

keywords such as `far`, `near`, or `huge` in the declaration of functions to be called from LabWindows/CVI source code.

- If your Watcom-compiled module performs floating point operations, you must use Watcom Version 9.5 or later.
- Use the following options when you compile with Watcom IDE:
  - Set the Project Target Environment to 32-bit Windows 3.*x*, and set the Image Type to Library [.lib].
  - Turn on the Disable Stack Depth Checking [-s] option.
  - Turn on the Change Char Default to Signed [-j] option.
  - Add `-zw -d_NI_mswin16_` to the **Other Options.**
  - Turn on the Generate as Needed [-of] option for Stack Frames.
  - Turn on the No Debugging Information option.
  - Turn on the In-line with Coprocessor [fpi87] option for Floating Point Model.
  - Turn on the Compiler default option for the Memory Model.
  - Turn on the 80486 Stack-Based Calling [-4s] option for the Target Processor.
- Use the following compiler flags when using `wcc386` or `wcc386p`:
  - `-zw -s -4s -j -fpi87 -d0 -of -d_NI_mswin16_`
  - You can use optimization flags in addition to the `f`, and you can use other flags, such as `-wn`, which do not affect the generation of object code.

Using 32-Bit Borland or Symantec Compiled Modules under Windows 3.1

In this section, *CVI* refers to both LabWindows/CVI and Watcom modules, while *Borland* applies to both Borland and Symantec modules.

The following restrictions apply to Borland object modules:

- Borland packs bit fields in structures differently than CVI, so you cannot share structures with bit fields between Borland and CVI.
- Borland returns structures, floats, and doubles differently than CVI. Therefore, functions that return these types cannot be called from CVI if they are defined in Borland, or vice versa. The exceptions are the ANSI C library functions that return doubles, which you can call from within Borland compiled modules.

Note ***This rule applies only to return values. You can use structs, floats and doubles as output parameters without limitation.***

- ANSI C library functions `div` and `ldiv` return structures, and hence you cannot call them from Borland compiled modules.
- The type `long double` is the same as `double` in CVI, while in Borland it is 10 bytes long, so you cannot share objects of this type between Borland and CVI modules. This affects the `"%Le"`, `"%Lf"`, `"%Lg"` format specifiers of `printf`, `sprintf`, `fprintf`, `scanf`, `sscanf`, `fscanf`, and others.
- Because you cannot share structures with bit fields between Borland and CVI, you cannot use the macros in `stdio.h` (`getc`, `putc`, `fgetc`, `fputc`) in Borland objects.
- `wchar_t` is defined as a `char` in CVI, whereas it is defined as a `short` in Borland, so ANSI C library functions that return `wchar_t` or take `wchar_t` parameters do not work.

Use the following options when you compile with Borland C 4.*x*:

- Set the target to be a Win32 application.
- Define `_NI_mswin16_`.
- Set the include directories to point to `cvi\include` before other include directories.
- Turn off the Allocate Enums as Ints option.
- Turn off the Fast Floating Point option.
- Use the C calling convention.

If you use a file with a `.c` extension, Borland C++ 4.*x* compiles it as a C source file. If your file has a `.cpp` extension, Borland C++ 4.*x* compiles it as a C++ source file; you must use `extern "C"` for any functions or variables you want to access from a C file.

Use the following options when you compile with Symantec C++ 6.0:

- Set the target to be a Win32s executable.
- Define `_NI_mswin16_`.
- Set the include directories to point to `cvi\include` before any other include directories.
- Set Structure Alignment to 1 byte.
- Turn off the Use Pascal Calling Convention option.

16-Bit Windows DLLs

You can call functions in a 16-bit DLL from source code or from a 32-bit compiled module. You can compile your 16-bit DLL in any language using any compiler that generates DLLs. If you want to program with DMA or interrupts, or access the Windows API, you *must* use a Windows DLL.

You must observe certain rules and restrictions in a DLL you want to use with LabWindows/CVI. If you experience problems using a DLL in LabWindows/CVI, you might have to contact the developer of the DLL to obtain modifications.

Because LabWindows/CVI is a 32-bit application, special *glue code* is required to communicate with a 16-bit DLL. For some DLLs, LabWindows/CVI can automatically generate this glue code from the include file when loading the DLL. For other DLLs, you have to modify the glue source code and compile it with Watcom into a `.obj` or `.lib` file.

The normal way of communicating with a DLL is by calling functions in the DLL. However, cases exist where you must use other communication methods. The most typical case is that of an interrupt service routine in a DLL that notifies the application when an interrupt occurs. This is done through a callback function. Also, LabWindows/CVI can recognize messages posted by a DLL through the Windows Application Programming Interface (API) function `PostMessage` and initiate a callback function.

Helpful LabWindows/CVI Options for Working with DLLs

LabWindows/CVI provides two options that can be helpful when working with DLLs. The options can be found in the **Run Options** menu of the Project window:

- Enable the Check Disk Dates Before Each Run option when you iteratively modify a DLL or DLL glue code file and run a LabWindows/CVI test program that calls into the DLL. By enabling the Check Disk Dates Before Each Run option, you ensure that you link the most recent version of the DLL and DLL glue code into your program. You can leave this option enabled at all times. The only penalty is a small delay each time you build or run the project.
- By default, LabWindows/CVI does not unload and reload DLLs between each execution of your program. This eliminates the delay in reloading the DLLs before each run. It allows the DLLs to retain state information between each run. If, however, you use a DLL that does not work correctly across multiple program executions, enable the Reload DLLs Before Each Run option.

DLL Rules and Restrictions

To call into a 16-bit DLL from LabWindows/CVI 32-bit code, you must observe the following rules and restrictions for DLL functions:

- In the DLL header file, change all references to `int` into references to `short`.
- In the DLL header file, change all references to `unsigned` or `unsigned int` to `unsigned short`.
- You can declare the functions in the DLL as `PASCAL` or as `CDECL`.
- You cannot use variable argument functions.

- You can use the argument types `char`, `unsigned char`, `int`, `unsigned int`, `short`, `unsigned short`, `long`, `unsigned long`, `float`, and `double`, as well as pointers to any type, and arrays of any type. You can use typedefs for these types.
- You can use the return types `void`, `char`, `unsigned char`, `int`, `unsigned int`, `short`, `unsigned short`, `long`, and `unsigned long`, as well as pointers to any type. You can use typedefs for these types.
- You can use the return types float and double only if the DLL is created with a Microsoft C compiler, and the functions returning floats or double are declared with the `cdecl` calling convention. You do not have to modify the glue code generated for functions that return float or double values.
- In the DLL header file, enum sizes must be consistent between LabWindows/CVI and the compiler for the DLL.

  ```
  typedef enum {
  No_Error,
  Device_Busy,
  Device_Not_Found
  } ErrorType;
  ```

  The size of `ErrorType` is 2 bytes in Visual C++, whereas it is 1 byte in LabWindows/CVI. To force LabWindows/CVI to treat `ErrorType` as 2 bytes, add another enum value explicitly initialized to a 2-byte value, such as the following.

  ```
  ErrorType_Dummy = 32767
  ```

- If the DLL you are using performs DMA on a buffer you pass to it, you might experience a problem. The DLL might attempt to lock the buffer in memory by calling the Windows SDK function `GlobalPageLock`. `GlobalPageLock` fails on buffers allocated with the Watcom `malloc` function that LabWindows/CVI uses in 32-bit mode.

  Write the DLL so that if `GlobalPageLock` fails, the DLL attempts to lock the buffer with the following code:

  ```
  int DPMILock (void *buffer, unsigned long size)
  {
      DWORD base;
      unsigned sel, offset;
      union _REGS regs;
      sel = SELECTOROF(buffer);
      offset = OFFSETOF(buffer);
      base = GetSelectorBase(sel);
      base = base+offset;

      regs.x.ax = 0x600; /* DPMI lock memory function */
      regs.x.bx = HIWORD(base);
      regs.x.cx = LOWORD(base);
      regs.x.di = LOWORD(size);
  ```

```
    regs.x.si = HIWORD(size);
    int86(0x31, &regs, &regs);
    return regs.x.cflag;
}
```

After the DMA is complete, you must unlock the buffer. You can unlock the buffer using the `DPMILock` function, if you set `regs.x.ax` to `0x601`, instead of `0x600`.

- If you compile the DLL with the `/FPi` or `/FPc` switches or with no `/FP` switches (`/FPi` is the default), the DLL uses the `WIN87EM.DLL` floating point emulator. LabWindows/CVI does not use `WIN87EM.DLL`. If the DLL uses `WIN87EM.DLL`, use the following strategy in the DLL to prevent conflicts:
  1. Structure the code so that all functions that perform any floating-point math have known entry and exit points. Ideally, specify a particular set of exported entry points as the only ways into the floating-point code.
  2. Call the Windows SDK function `FPInit` in each of these entry points. Store the previous signal handler in a function pointer.
  3. If the DLL has its own exception handler, call `signal` to register the DLL's own signal handler.
  4. Perform the floating-point math.
  5. Upon exiting through one of the well-defined DLL exit points, call the Windows SDK function `FPTerm` to restore the previous exception handler and terminate the DLL's use of `WIN87EM.DLL`.

```
typedef void (*LPFNSIGNALPROC) (int, int);

/* prototypes for functions in WIN87EM.d11 */
LPFNSIGNALPROC PASCAL_FPInit (void);
VOID PASCAL_FPTerm (LPFNSIGNALPROC);

void DllFunction (void)
{
    LPFNSIGNALPROC  OldFPHandler;

    /* save the floating point state, and setup the  */
    /* floating point exception handler for this DLL.*/
    OldFPHandler = _FPInit ();
    signal ( SIGFPE, DLLsFPEHandler); /* optional    */
    .
    .
    .
```

```
    /* perform the computations                          */
    .
    .
    .
    /* restore the floating point state                  */
    _FPTerm (OldFPHandler);
}
```

☞ **Note** ***If you use Microsoft C to build the DLL, you might get a linker error for an undefined symbol*** `_acrtused2`***. This error occurs only in Microsoft C versions 7.00 and later. Include the following dummy function in your DLL to fix this error. Also, when linking to the DLL, specify*** `WIN87EM.LIB` ***as the first library to be linked.***

```
void _acrtused2 (void)
{
}
```

DLL Glue Code

Because LabWindows/CVI is a 32-bit application, it does not use 16-bit import libraries or import statements in module definition files. Instead, LabWindows/CVI uses 32-bit DLL glue code. In some cases, it is sufficient to use glue code that LabWindows/CVI automatically generates when it loads the DLL. However, you *cannot* use this method in the following cases:

- The DLL requires special interface functions compiled outside of the DLL.
- You expect to pass arrays bigger than 64 K to functions in the DLL.
- You pass a pointer to a function in the DLL, and the DLL uses the pointer after the function returns. For example, you pass an array to a function that starts an asynchronous I/O operation. The function returns immediately, but the DLL continues to operate on the array.
- You pass a function pointer to the DLL, and the DLL calls the function later. For example, the DLL makes a direct callback into 32-bit code.
- You pass to the DLL a pointer that points to other pointers. Two examples of pointers that point to other pointers are an array of pointers and a structure pointer with pointer members.
- The DLL returns pointers as return values or through reference parameters.
- The DLL exports functions by ordinal value only.

If your DLL falls into any of these categories, refer to the *DLLs That Cannot Use Glue Code Generated at Load Time* section of this chapter for details on how to proceed. Otherwise, refer to the *DLLs That Can Use Glue Code Generated at Load Time* section, also in this chapter.

DLLs That Can Use Glue Code Generated at Load Time

If your DLL can use glue code generated at load time, LabWindows/CVI automatically generates the glue code based on the contents of the `.h` file it associates with the DLL when it loads it.

Any functions declared as `PASCAL`, `pascal`, or `_pascal` in the DLL should be declared as `PASCAL` in the `.h` file. LabWindows/CVI ignores the `PASCAL` keyword except when generating the glue code.

Use only standard ANSI C keywords in the `.h` file. (The keyword `PASCAL` is the only exception to this rule.) For example, do not use `far`, `near`, or `huge`.

☞ **Note** ***You can create an object module that contains the glue code. If you do so, LabWindows/CVI can load the DLL faster because it does not have to regenerate and recompile the glue code. To create the object module, load the*** `.h` ***file into a Source window and select*** **Options»Generate DLL Glue Object.** ***If the DLL pathname is listed in the project, replace it with the object module file. If the DLL is not listed in the project, but is associated with a*** `.fp` ***file, make sure the object module is in the same directory as the*** `.fp` ***file.***

DLLs That Cannot Use Glue Code Generated at Load Time

If your DLL cannot use glue code generated at load time, you must generate a glue code source file from the DLL include file using the **Generate DLL Glue Source** command from the **Options** menu of a Source window. You must then compile the glue code using the Watcom compiler to create a `.obj` or `.lib` file to be loaded with the DLL. If you also have interface functions that must exist outside the DLL, you must combine them with the glue code to form the `.obj` or `.lib` file.

Loading a DLL That Cannot Use Glue Code Generated at Load Time

If you have a 32-bit Watcom compiled `.obj` or `.lib` file that contains glue code for a DLL, LabWindows/CVI must load the `.obj` or `.lib` file first. For instance, if you want to use `x.dll` and `x.obj` in your program, add `x.obj` to the project. Do *not* add `x.dll` to the project. The `.obj` or `.lib` file causes LabWindows/CVI to load the `.dll`.

The `.obj` or `.lib` file must contain the glue code for the DLL. It is the presence of the glue code that indicates to LabWindows/CVI that a `.dll` is associated with the `.obj` or `.lib` file.

When LabWindows/CVI loads the `.obj` or `.lib` file and finds that it contains glue code, it first looks for the `.dll` in the same directory as the `.obj` or `.lib` file. If it cannot find the `.dll`, LabWindows/CVI looks for it using the standard Windows DLL search algorithm.

Also, you can create a .pth file in the same directory as the .obj or .lib file with the same base name. The .pth file must contain a simple filename or a full pathname of the DLL. If it is a simple filename, LabWindows/CVI uses the standard Windows DLL search algorithm.

Rules for the DLL Include File Used to Generate Glue Source

You can generate the DLL glue source file by opening the .h file for the DLL in a Source window and selecting **Generate DLL Glue Source** from the **Options** menu. This command prompts you for the name of a .h file. It puts the glue code in a .c file with the same path and base name as the .h file. You must modify this .c file as this section describes and compile it using the Watcom compiler. Refer to the *Using 32-Bit Watcom Compiled Modules under Windows 3.1* section of this chapter for information on how to use the Watcom compiler with LabWindows/CVI.

If any of the functions in the DLL are declared as PASCAL, pascal, or _pascal, you must declare them as PASCAL in the .h file you use to generate the glue code. LabWindows/CVI ignores the PASCAL keyword except for the purposes of generating the glue code. The stub function in the glue code is *not* declared as PASCAL. If you include this .h file in the glue code, the Watcom compiler flags as an error the inconsistency between the declaration of the function in the .h file and the definition of the stub function. If you include it in other modules you compile under Watcom, calls to the function erroneously compile as if the function were PASCAL. You have two options:

- Have two separate .h files, one that includes the PASCAL keyword and one that does not. Use the one that does include the PASCAL keyword to generate the glue code only.
- Use conditional compilation so that Watcom ignores the PASCAL macro when it compiles.

Only use standard ANSI C keywords in the .h file. The keyword PASCAL is the only exception to this rule. For example, do not use far, near, or huge.

If the DLL Requires a Support Module outside the DLL

Support modules contain special interface functions that the DLL uses but that exist outside of the DLL. If you are unsure whether the DLL requires a support module, try to build a project in LabWindows/CVI with the DLL in the project list. If link errors exist in the form of unresolved references, the DLL requires special interface functions. Get the source code for the interface functions, add it to the glue code, and compile using the Watcom compiler.

If You Pass Arrays Bigger Than 64 K to the DLL

If you pass the DLL any arrays bigger than 64 K, you must modify the glue code source file. For example, suppose you have a function in the DLL with the following prototype:

```
long WriteRealArray (double realArray[], long numElems);
```

In the glue code generated by LabWindows/CVI, there is a declaration of `WriteRealArray` like that shown in the following example.

```
long WriteRealArray (double realArray[], long numElems)
{
    long retval;
    unsigned short cw387;
    cw387 = Get387CW();
    retval = (long)
    InvokeIndirectFunction (__static_WriteRealArray, realArray,
                            numElems);
    Set387CW (cw387);
    return retval;
}
```

Note ***The lines of code referencing*** `cw387` ***are necessary only if the DLL function performs floating point operations. They are innocuous and execute quickly, so LabWindows/CVI adds them to the glue code automatically. If the DLL function does not perform floating point operations, you can remove these lines.***

If `realArray` can be greater than 64 K, you must modify the interface routine as shown.

```
long WriteRealArray (double realArray[], long numElems)
{
    long retval;
    unsigned short cw387;
    DWORD size;
    DWORD alias;

    size = numElems * sizeof(double);
    if (Alloc16BitAlias (realArray, size, &alias) <0)
       return <error code>;
    cw387 = Get387CW();
    retval = (long)
    InvokeIndirectFunction (__static_WriteRealArray, alias,
                            numElems);
    Set387CW (cw387);
    Free16BitAlias (alias, size);
    return retval;
}
```

You must also modify the call to `GetIndirectFunctionHandle` for `WriteRealArray` as shown in the following code:

```
if (!(__static_WriteRealArray = GetIndirectFunctionHandle
     (fp,  INDIR_PTR, INDIR_WORD, INDIR_ENDLIST)))
```

by changing `INDIR_PTR` to `INDIR_DWORD`.

If the DLL Retains a Buffer after the Function Returns (an Asynchronous Operation)

If the DLL retains a buffer after the function returns, you must modify the glue code source file. Suppose two functions exist. `WriteRealArrayAsync` operates just like `WriteRealArray`, except that it returns before it completes writing the real array. `ClearAsyncWrite` terminates the asynchronous I/O. The glue code interface functions for `WriteRealArrayAsync` and `ClearAsyncWrite` should be modified to resemble the following example.

```
static DWORD gAsyncWriteAlias, gAsyncWriteSize;

long WriteRealArrayAsync (double realArray[], long numElems)
{
    long retval;
    unsigned short cw387;
    DWORD size;
    DWORD alias;

    size = numElems * sizeof(double);
    if (Alloc16BitAlias (realArray, size, &alias) < 0)
        return <error code>;
    cw387 = Get387CW();
    retval = (long)
    InvokeIndirectFunction (__static_WriteRealArrayAsync, alias,
                            numElems);
    Set387CW (cw387);
    if (IsError (retval))  /* replace with macro to check if */
                           /* retval is error               */
       Free16BitAlias (alias, size);
    else {
       gAsyncWriteAlias = alias;
       gAsyncWriteSize = size;
    }
    return retval;
}
```

```
long ClearAsyncWrite (void)
{
   /* because this does no floating point, you can remove */
   /* the cw387 code                                      */
   long retval;

   retval = (long) InvokeIndirectFunction(__static_ClearAsyncWrite);
   if (!IsError (retval))  /* replace with macro to check if  */
                           /* retval is error                 */
      if (gAsyncWriteAlias != 0) {
         Free16BitAlias (gAsyncWriteAlias,gAsyncWriteSize);
         gAsyncWriteAlias = 0;
         gAsyncWriteSize = 0;
      }
   return retval;
}
```

You can terminate LabWindows/CVI programs in the middle of execution and then re-run them. When you terminate the program, you should also terminate the asynchronous I/O. You can arrange to be notified of changes in the run state by including a function with the name `RunStateChangeCallback` in the `.obj` or `.lib` file associated with the DLL. You can add this function to the glue code file. Refer to the *Notification of Changes in Run State* section of Chapter 2, *Using Loadable Compiled Modules*, of this manual for a complete description of the run state change notification. In the example we have been discussing, you should add the following code.

```
#include "libsupp.h"
void CVICALLBACK __RunStateChangeCallback (int newState)
{
   if (newState == kRunState_Stop)
       ClearAsyncWrite ();
}
```

If the DLL Calls Directly Back into 32-Bit Code

If the DLL calls directly back into 32-bit code, you must modify the glue code source file. You can call functions defined in 32-bit source code directly from a DLL. Although this method is not as straightforward as Windows messaging, it is not subject to the latencies of Window messaging. For more information about Windows messaging, refer to the *Recognizing Windows Messages Passed from a DLL* section of this chapter.

Note ***If you need direct callbacks to occur at interrupt time because the latency of Windows messaging is interfering with your application, contact National Instruments for assistance.***

You cannot pass pointers to 32-bit functions directly into 16-bit DLLs. The Windows SDK interface for this is very complex. **Generate DLL Glue Source** does not generate this code for you. You must write your own glue code for passing function pointers to and from a DLL, and add it to the file that **Generate DLL Glue Source** generates.

Suppose a DLL contains the following functions:

```
long (FAR*savedCallbackPtr) (long);
long  FAR InstallCallback(long (FAR*callbackPtr) (long))
{
   savedCallbackPtr = callbackPtr;
}
long InvokeCallback(long data)
{
   return (*savedCallbackPtr)(data);
}
```

After you use the **Generate DLL Glue Source** command to generate the glue code for these functions, you must modify the code as follows.

Note ***Because direct callbacks must be declared*** `far`***, and LabWindows/CVI cannot compile*** `far` ***functions, you must declare a*** `far` ***function in the glue code and pass it to the DLL. This*** `far` ***function calls the actual user function.***

```
#undef MakeProcInstance    /* Use version that does not  */
                           /* convert pointer.           */
#undef FreeProcInstance    /* Use version that does not  */
                           /* convert pointer.           */

typedef struct {  /* Holds resources required to register*/
                  /* the callback.                       */
   int UserDefinedProcHandle;
   CALLBACKPTR proc16;
   FARPROC proc16Instance;
}  CallbackDataType;
static CallbackDataType CallbackData;

static long (*UsersCallback)(long);

/* Define a 32-bit far callback whose address is passed to */
/* the DLL. It calls your function using function pointer  */
/* stored in UsersCallback.                                */
```

```
static long FAR CallbackHelper(long data)
{
   return  (*UsersCallback)(data);
}

/* Modified glue code for the function that installs the   */
/* callback.                                               */
long InstallCallback(long (*callback)(long))
{
   long retval;
   unsigned short cw387;

   UsersCallback = callback; /* Store CVI 32-bit pointer  */
                             /* in static variable.       */

   /* Create a 16-bit thunk for the 32-bit far function   */
   /* CallbackHelper                                      */

   if ((CallbackData.UserDefinedProcHandle =
                           GetProcUserDefinedHandle()) == 0)
      return FALSE;/* Too many callbacks installed        */
                  /* or handles not freed.                */

   if (DefineUserProc16(CallbackData.UserDefinedProcHandle,
                        (PROCPTR) CallbackHelper, UDP16_DWORD,
                        UDP16_CDECL, UDP16_ENDLIST))
      goto failed;

   if (!(CallbackData.proc16 =
            GetProc16((PROCPTR)CallbackHelper,
                      CallbackData.UserDefinedProcHandle)))
      goto failed;

   CallbackData.proc16Instance =
            MakeProcInstance(CallbackData.proc16,
                             GetTaskInstance());

   cw387 = Get387CW();
   retval = (long)
   InvokeIndirectFunction(__static_InstallCallback,
                          CallbackData.proc16Instance);
   Set387CW(cw387);
   return retval;
```

```
failed:
    FreeCallbackResources();
    return FALSE;
}

/* Call this function after unregistering the callback. */
void FreeCallbackResources(void)
{
    if (CallbackData.proc16Instance) {
        FreeProcInstance(CallbackData.proc16Instance);
        CallbackData.proc16Instance = 0;
    }
    if (CallbackData.proc16) {
        ReleaseProc16(CallbackData.proc16);
        CallbackData.proc16 = 0;
    }
    if (CallbackData.UserDefinedProcHandle) {
        FreeProcUserDefinedHandle(CallbackData.UserDefinedProcHandle);
        CallbackData.UserDefinedProcHandle = 0;
    }
}
```

If the DLL Returns Pointers

DLLs return pointers that fall into the following two classes.

- Pointers to memory that LabWindows/CVI allocates, that you pass into the DLL, and that the DLL later returns

  You must map these pointers back into normal 32-bit pointers that you can use in LabWindows/CVI code. Use the function `MapAliasToFlat` to convert these pointers.

- Pointers to memory that a DLL allocates

  Because these pointers point to memory that is not in the LabWindows/CVI flat address space, you cannot map them back into the normal 32-bit pointers that LabWindows/CVI uses. You can access them in Watcom object code by first converting them to 32-bit far pointers using the function `MK_FP32`.

  To access them in LabWindows/CVI source code you must copy the data into a buffer you allocate in LabWindows/CVI. Notice that you cannot pass 16- or 32-bit far pointers to LabWindows/CVI library functions, and that LabWindows/CVI does not provide access to the Watcom string and memory buffer manipulation functions that take far pointers as arguments. You must write the loops to copy the data.

Case 1

Assume the DLL has the following function:

```
char *f(char *ptr)
{
   sprintf(ptr, "hello");
   return ptr;
}
```

Then assume that a program in LabWindows/CVI uses the function `f` as follows:

```
char buffer[240];
char *bufptr;
bufptr = f(buffer);
printf("%s", bufptr);
```

You would have to modify the glue code as shown here:

```
char * f(char *ptr)
{
   char * retval;
   unsigned short cw387;

   cw387 = Get387CW();
   retval = (char *) InvokeIndirectFunction(__static_f, ptr);
   Set387CW(cw387);
   retval = MapAliasToFlat(retval); /* Add this line to */
                                    /* glue code.       */
   return retval;
}
```

Case 2

Assume the DLL has the following function:

```
char *f(void)
{
   char *ptr;

   ptr = malloc(100);
   sprintf(ptr, "hello");
   return ptr;
}
```

Then assume that a program in LabWindows/CVI uses the function `f` as follows:

```
char *bufptr;
bufptr = f();
printf("%s", bufptr);
```

You would have to modify the glue code as shown here:

```
char * f(char *ptr)
{
    char *retval;
    unsigned short  cw387;
    char *ptr, *tmpPtr, _far *farPtr32, _far *tmpFarPtr32;
    int i;

    cw387 = Get387CW();
    retval = (char *) InvokeIndirectFunction(__static_f, ptr);
    Set387CW(cw387);

    /* convert the 16 bit far pointer to a 32 bit far pointer*/
       farPtr32 = MK_FP32(retval);
    tmpFarPtr32 = farPtr32;

    /* Calculate the length of the string. Cannot call strlen*/
    /* because it does not accept far pointers.              */

    i = 0
    while (*tmpFarPtr32++)
       i++;

    /* Allocate buffer from CVI memory and copy in data.     */
    if ((ptr = malloc(i + 1)) != NULL) {
       tmpFarPtr32 = farPtr32;
       tmpPtr = ptr;
       while (*tmpPtr++ = *tmpFarPtr32++);
    }
    return ptr;
}
```

If a DLL Receives a Pointer that Points to Other Pointers

Assume the following DLL functions:

```
int f(char *ptrs[]);
struct x {
    char *name;
};
int g(struct x *ptr);
```

For the function `f`, the glue code that LabWindows/CVI generates converts the pointer to the array `ptrs` to a 16-bit far pointer when you pass it to the DLL function, but does not convert the pointers inside the array (`ptrs[0], ptrs[1], ...`). Similarly, for the function `g`, the glue code that LabWindows/CVI generates converts the pointer to the structure (`ptr`), but not the pointer inside the structure (`name`).

If your DLL has functions with these types of parameters, then your DLL cannot use glue code automatically generated at load time. You can use the **Generate DLL Glue Source** command to generate glue code and then modify it in the following manner.

1. Before the call to `InvokeIndirectFunction`,
   a. Save the hidden pointer in a local variable.
   b. Replace the hidden pointer with a 16-bit alias by calling `Alloc16BitAlias`.
2. After the call to `InvokeIndirectFunction`,
   a. Free the 16-bit alias by calling `Free16BitAlias`.
   b. Restore the hidden pointer with the value you saved in step 1.

For the functions `f` and `g`, the glue code that LabWindows/CVI generates looks like the following excerpt:

```
int f(char **ptrs)
{
    int retval;
    unsigned short cw387;

    cw387 = Get387CW();
    retval = (int) InvokeIndirectFunction(__static_f, ptrs);
    Set387CW(cw387);
    return retval;
}
int g(struct x *ptr)
{
    int retval;
    unsigned short cw387;
```

```
    cw387 = Get387CW();
    retval = (int) InvokeIndirectFunction(__static_g, ptr);
    Set387CW(cw387);
    return retval;
}
```

After you make the necessary changes, the code should appear as follows:

```
/* Assume NUM_ELEMENTS is the number of pointers in the input */
/* array. Assume ITEM_SIZE is the number of bytes pointed     */
/* to by each pointer. If you do not know ITEM_SIZE, but you  */
/* know that it is 64K or less, you can use 64K as ITEM_SIZE. */
int f(char **ptrs)
{
    int retval;
    unsigned short cw387;
    int i;
    char *savedPointers[NUM_ELEMENTS];

    /* change the pointers to 16-bit far pointers */
    for (i = 0 ; i < NUM_ELEMENTS; i++) {
        savedPointers[i] = ptrs[i];
        if (Alloc16BitAlias(ptrs[i], ITEM_SIZE, &ptrs[i]) == -1) {
            /* failed to allocate an alias; restore  */
            /* pointers.                             */
            while (i--)
                ptrs[i] = savedPointer[i];
            return <error code>;
        }
    }
    cw387 = Get387CW();
    retval = (int) InvokeIndirectFunction(__static_f, ptrs);
    Set387CW(cw387);

    /* Restore the pointers. */
    for (i = 0 ; i < NUM_ELEMENTS; i++) {
       Free16BitAlias(ptrs[i], ITEM_SIZE);
       ptrs[i] = savedPointers[i];
    }
    return retval;
}
```

```
int g(struct x *ptr)
{
   int retval;
   unsigned short cw387;
   char *savedPointer;

   savedPointer = ptr->name;
   if (Alloc16BitAlias(ptr->name, ITEM_SIZE, &ptr->name) == -1)
      return <error code>;

   cw387 = Get387CW();
   retval = (int) InvokeIndirectFunction(__static_g, ptr);
   Set387CW(cw387);

   Free16BitAlias(ptr->name, ITEM_SIZE);
   ptr->name = savedPointer;
   return retval;
}
```

DLL Exports Functions by Ordinal Value Only

If your DLL does not export its functions by name, but by ordinal number only, you must modify the `GetProcAddress` function calls in the glue code. Instead of passing the name of the function as the second parameter, pass `PASS_WORD_AS_POINTER(`*`OrdinalNumber`*`)`, where *`OrdinalNumber`* is the ordinal number for the function. For example, if the ordinal number for the function `InstallCallback` is 5, change the glue code as follows.

Generated Glue Code:

```
if (!(fp = GetProcAddress(DLLHandle,"InstallCallback")))
{
   funcname = "_InstallCallback";
   goto FunctionNotFoundError;
}
```

Change to:

```
if (!(fp = GetProcAddress(DLLHandle, PASS_WORD_AS_POINTER(5))))
{
   funcname = "_InstallCallback";
   goto FunctionNotFoundError;
}
```

Recognizing Windows Messages Passed from a DLL

The normal way of communicating with a DLL is to call functions in the DLL. However, cases exist where other communication methods are necessary. The most typical case is that of an interrupt service routine in a DLL that must notify the application that the interrupt occurred. In cases like this, you must communicate with the DLL through a callback function.

LabWindows/CVI recognizes messages posted by a DLL through the Windows SDK function `PostMessage`, and can initiate a user callback function. This method is useful for hardware interrupts, but it is subject to the latency associated with Windows messaging. LabWindows/CVI uses `RegisterWinMsgCallback`, `UnRegisterWinMsgCallback`, and `GetCVIWindowHandle` to recognize Windows messages from a DLL. You can call these functions from a module compiled in Watcom or from source code.

For complete information on these functions, refer to the function descriptions in Chapter 4, *User Interface Library Reference*, of the *LabWindows/CVI User Interface Reference Manual*.

To use these functions, call `RegisterWinMsgCallback` and `GetCVIWindowHandle`. Pass their return values, the message number and the window handle, to the DLL. When the DLL sends a message, it calls `PostMessage` with these values. When LabWindows/CVI receives the message, it calls the callback function.

☞ **Note** ***LabWindows/CVI can receive the message only when it is processing events. LabWindows/CVI processes events when it is waiting for user input. If the program you run in LabWindows/CVI does not call*** `RunUserInterface`***,*** `GetUserEvent`***, or*** `scanf`***, or if it does not return from a User Interface Library callback, events will not be processed. You can remedy this in the program by periodically calling the User Interface Library function*** `ProcessSystemEvents`**.**

Creating 16-bit DLLs with Microsoft Visual C++ 1.5

Be sure to consider the following issues or project options when you create a DLL with Microsoft Visual C++ 1.5:

- Every function you call from outside the DLL must be `far`, exported, and must load the data segment into the DS register. The function must load the DS register if you want to use any non-local variables in a function.
- Use the large or huge memory model. The savings you gain by using smaller memory models is not worth having to use the `far` keyword throughout your code. This project option is in **Compiler»Memory Model»Segment Setup**.
- You can make the compiler load the data segment into the DS register automatically by using the **SS!=DS, DS loaded on function entry** project option in **Compiler»Memory Model»Segment Setup**.
- If you try to use the optimize entry code option (`/GD`), by selecting **Compiler»Windows»Prolog/Epilog»Generate Prolog/Epilog For**, it conflicts with the

/Au option. You can either not use this option by setting it to **None**, or insert __loadds in front of every function you export from the DLL.

- You can make the compiler export a function by inserting __export between the return type and the function name, or by adding the function name to the exports section of the .def file.
- If you add the function name to the exports section of the .def file, remember to convert the name to all caps if you use the PASCAL calling convention, or pre-append an underscore if you use the CDECL calling convention.
- Byte align structure members by choosing **1 Byte** for the **Options»Project»Compiler» Code Generation»Struct Member Byte Alignment**.

Creating 16-bit DLLs with Borland C++

Consider the following issues or project options when you create a DLL with Borland C++ 4.*x*:

- Every function you call from outside the DLL must be far, exported, and must load the data segment into the DS register. The function must load the DS register if you want to use any non-local variables in a function.
- Use the large or huge memory model. The savings you gain by using smaller memory models is not worth having to use the far keyword throughout your code. This project option is in **16-bit Compiler»Memory Model»Mixed Model Override**.
- You can make the compiler load the data segment into the DS register by setting the project option **16-bit Compiler»Memory Model»Assume SS Equals DS** to **Never**, or by inserting _loadds in front of every function you export from the DLL.
- You can make the compiler export a function by inserting _export between the return type and the function name, adding the function name to the exports section of the .def file, or setting the option **16-bit Compiler»Entry/Exit Code»Windows DLL, all functions exportable**.
- If you add the function name to the exports section of the .def file, remember to convert the name to all caps if you use the PASCAL calling convention, or pre-append an underscore if you use the CDECL calling convention. Also, set the Generate Underscores option in **Compiler»Compiler Output**.
- Turn off the Allocate Enums as Ints option in **Compiler»Code Generation**.
- Set the Data Alignment options to **Byte** in **16-bit Compiler»Processor.**
- Turn off the Case Sensitive Link and Case Sensitive Exports and Imports options in the **Linker»General**.
- Do not use the Linker Goodies options in **Linker»16-bit Linker**.

DLL Search Precedence

LabWindows/CVI finds a DLL file in the following ways for Windows 3.1:

- If the `.dll` file is associated with a `.fp` file, LabWindows/CVI uses the following search precedence to find the DLL.
  1. If a `.pth` file with the same *full* path name as the `.fp` file is in the project, LabWindows/CVI uses the standard Windows DLL search algorithm. The `.pth` file must contain the name of the `.dll` file, such as `mystuff.dll`. It must contain an absolute path or a simple filename.
  2. If a `.dll` file with the same *full* path name as the `.fp` file is in the project, LabWindows/CVI uses the absolute path of the `.dll` file in the project to load the `.dll` file.
  3. If a `.pth` file with the same base name as the `.fp` file is in the same directory as the `.fp` file and a `.lib` or `.obj` file of the same base name does not exist in the same directory, LabWindows/CVI uses the standard Windows DLL search algorithm. The `.pth` file must contain the name of the `.dll` file, such as `mystuff.dll`. It must not contain any directory names or slashes.
  4. If a `.dll` file with the same base name as the `.fp` file is in the same directory as the `.fp` file, LabWindows/CVI loads the `.dll` file as long as no `.lib`, `.obj`, or `.pth` file of the same base name appears in the same directory.
  5. If a `.pth` or `.dll` file does not appear in the same directory as the `.fp` file, LabWindows/CVI uses the standard Windows search algorithm to look for a DLL with the same base name as the `.fp` file. Thus, if a DLL with the same base name is in the `windows` or `windows\system` directory or a directory listed in your `PATH` environment variable, LabWindows/CVI finds it.

  DLLs for VXI*plug&play* drivers are not in the same directory as the `.fp` files, but the directory that contains the DLL is listed in the `PATH` environment variable. Therefore, Step 5 makes it easier for you to use VXI*plug&play* instrument driver DLLs in LabWindows/CVI for Windows 3.1.
- If the `.dll` file is not associated with a `.fp` file, LabWindows/CVI uses the following search precedence to find the DLL:
  1. If a `.pth` file is in the project list, LabWindows/CVI uses the standard Windows DLL search algorithm. The `.pth` file must contain the name of the `.dll` file, such as `mystuff.dll`. It must contain an absolute path or a simple filename.
  2. If the `.dll` file is in the project list, then LabWindows/CVI uses the absolute pathname to find the `.dll` file.

- If you call `LoadExternalModule` on the `.dll` file, then
  - If you specify it with an absolute pathname, LabWindows/CVI loads that file.
  - If you specify it with a relative pathname, LabWindows/CVI searches for the `.dll` file in the following places and order indicated.
    1. In the project list.
    2. In the directory in which the project file is located.
    3. Among other modules already loaded.
    4. In the directories specified in the documentation for the Windows SDK `LoadLibrary` function. In this case, the include file for the DLL must be in the project or in one of the include paths you specify in the **Include Paths** command in the **Options** menu of the Project window.

5

UNIX Compiler/Linker Issues

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

Calling Sun C Library Functions

You can call functions in the Sun Solaris C libraries from source code in LabWindows/CVI. LabWindows/CVI automatically links your program to the following static libraries, located in the `/usr/lib` directory, when you build the project.

Solaris 1: `libm.a, libc.a`

Solaris 2: `libsocket.a, libnsl.a, libintl.a, libm.a, libc.a`

When you create a standalone executable, LabWindows/CVI invokes the Sun Solaris link editor (`ld`) to link your program to the LabWindows/CVI dynamic library and to the system libraries. By default, the Sun Solaris link editor uses the dynamic versions of the libraries. LabWindows/CVI passes the following linking options to the Sun Solaris link editor:

Solaris 1: `-lm -ldl -lc`

Solaris 2: `-lsocket -lnsl -lintl -lm -lthread -lc`

In general, you can use the header files that Sun provides for these libraries in the `/usr/include` directory. For the ANSI C functions, however, use the header files that come with LabWindows/CVI.

Restrictions on Calling Sun C Library Functions

You cannot call any Sun C Library function that uses data types incompatible with the LabWindows/CVI compiler or libraries. In particular, you must not call functions that use the `long double` data type. In LabWindows/CVI the `long double` data type has 8 bytes, but the Sun libraries expect a 16-byte object.

Under Solaris 2, you must not call any function that uses the `long long` data type. LabWindows/CVI does not recognize this non-ANSI type.

Using Shared Libraries in LabWindows/CVI

In the LabWindows/CVI development environment, you can link your programs to static libraries, but not to shared libraries. If you have to use a shared library, you must use the Sun Solaris linker (`ld`) to build your application. Refer to the *Creating Executables that Use the LabWindows/CVI Libraries* section later in this chapter for more information on using external compilers and the Sun linker.

If you have both shared and static versions of a library, you can develop and debug your application in the LabWindows/CVI development environment using the static version of the library. You can then create your final executable with the Sun linker using the shared version of the library.

Using dlopen

The Sun Solaris `dlopen` function allows you to load shared libraries from your program dynamically. Although this function can work in some cases when running in LabWindows/CVI, it can make LabWindows/CVI unstable. If you use `dlopen` to load shared libraries in a program you run in LabWindows/CVI, the shared libraries might link to the system libraries the LabWindows/CVI environment uses. As a result, functions in the shared library might modify the LabWindows/CVI environment and cause unpredictable behavior.

The LabWindows/CVI Run-Time Engine as a Shared Library

The LabWindows/CVI development environment contains many built-in libraries such as the User Interface Library and Utility Library. LabWindows/CVI also provides these libraries in the form of a standalone shared library called the LabWindows/CVI Run-time Engine. All executables that call LabWindows/CVI library functions use the Run-time Engine shared library. This is true whether you build the executable in the LabWindows/CVI development environment or with an external compiler and the Sun Solaris linker.

Creating Executables that Use the LabWindows/CVI Libraries

You can build executables that use the LabWindows/CVI libraries in two ways:

- You can build an executable in the LabWindows/CVI development environment by selecting the **Create Standalone Executable** command in the **Build** menu of the Project window. When you do so, LabWindows/CVI invokes the Sun Solaris linker (`ld`) to link your programs to the Run-time Engine shared library.
- You can use an external compiler and linker to create an executable that uses the Run-time Engine shared library. Use the Generate Makefile command in the **Build** menu of the Project window to generate a UNIX makefile that corresponds to the currently loaded project and libraries. The makefile invokes an external compiler to compile your source files, and then it invokes the Sun Solaris linker (`ld`) to link the compiled files with the Run-time Engine shared library.

Compatible External Compilers

You can use the following external ANSI C compilers to compile source files for linking with the LabWindows/CVI Run-time Engine shared library.

- GNU C Compiler (`gcc`)
- Sun C Compiler (`cc` and `acc`)

☞ **Note** ***Under Solaris 2.4, when linking the LabWindows/CVI Shared Library with external ANSI C compiler, the compiler displays a warning that states the shared library has an invalid type. You can ignore this warning.***

Static and Shared Versions of the ANSI C and Other Sun Libraries

When you build a project for execution in the LabWindows/CVI development environment, LabWindows/CVI links your program to the *static* versions of the Sun Solaris libraries (`libc.a` and `libm.a`). On the other hand, when you create a standalone executable in the LabWindows/CVI development environment, LabWindows/CVI invokes the Sun Solaris link editor (`ld`) to link your program to the *shared* versions of the libraries (`libc.so` and `libm.so`). Similarly, when you generate a UNIX makefile by invoking the **Generate Makefile** command from the **Build** Menu of the Project window, the makefile contains linker commands to use the shared versions of the libraries.

Thus, when you run your programs as executables, you use a different version of the Sun libraries (including the ANSI C library) than when you run them in the LabWindows/CVI development environment. Your program might exhibit slightly different behavior as a standalone executable than when run in the development environment.

Non-ANSI Behavior of Sun Solaris 1 ANSI C Library

The C library that comes with Sun Solaris 1 (SunOS 4.1.*x*) does not comply with the ANSI C standard as follows:

- Some ANSI C functions are missing from the library.
- Some library functions have different behavior than the ANSI standard specifies.

LabWindows/CVI corrects these problems by adding a library, `libcfix.a`, that replaces and supplements the Sun Solaris library as necessary. The *Solaris 1 ANSI C Library Implementation* section contains more information about how LabWindows/CVI provides an ANSI C library on Solaris 1.

LabWindows/CVI Implements printf and scanf

Although the Sun Solaris libraries provide the ANSI C family of functions for formatted input and output (`scanf`, `printf`, and others), LabWindows/CVI provides special versions of these functions for the following reasons:

- The LabWindows/CVI versions of these functions provide run-time error checking not available with Sun Solaris versions.
- The Sun Solaris 1 version of these functions do not comply fully with the ANSI C standard.
- The Sun Solaris versions of these functions do not work with the LabWindows/CVI implementation of the long double data type.

For standalone executables, these functions come in a separate static library, `libcviprintf.a`, in the `lib` subdirectory of the LabWindows/CVI installation directory. When you create an executable in LabWindows/CVI, LabWindows/CVI links your program to this static library.

Main Function Must Call InitCVIRTE

If your program calls any functions from the LabWindows/CVI libraries, you must call `InitCVIRTE` to initialize the libraries from the executable. This function takes three arguments. The first and third arguments to this function must always be 0 for UNIX applications. The second must be the same value as the second parameter of your `main` function. `InitCVIRTE` returns 0 if it fails.

You do not have to call `InitCVIRTE` when you run your program in the LabWindows/CVI development environment because LabWindows/CVI always initializes the libraries. However, if you do not call `InitCVIRTE`, your executable cannot work. For this reason,

National Instruments recommends that you always include source code similar to the following example in your program.

```
int main(int argc, char *argv[])
{
    if (InitCVIRTE(0, argv, 0) == 0) {
        return 1;/* Failed to initialize */
    }
    /* your program code here */
}
```

If you pass `NULL` for the second argument to `InitCVIRTE`, your program might still work, but with the following limitations:

- Your executable cannot accept the `-display` command line argument. As a result, you cannot specify an X display on the command line for your program to use. You still can use the `DISPLAY` environment variable to specify a different X display.
- `LoadPanel`, `LoadExternalModule`, `DisplayImageFile`, `SavePanelState`, `RecallPanelState`, and other functions that normally use the directory of the executable to search for files, use the current working directory instead. If you run the executable from a directory other than the one that contains your executable, some of these functions might fail to find files.

Run State Change Callbacks Are Not Available in Executables

When you use a compiled module in LabWindows/CVI, you can arrange for LabWindows/CVI to notify it of a change in execution status (start, stop, suspend, resume). You do this through a function called `__RunStateChangeCallback`. The *Notification of Changes in Run State* section, in Chapter 2, *Using Loadable Compiled Modules*, describes this in detail.

The run state change callback capability in LabWindows/CVI is necessary because when you run a program in the LabWindows/CVI development environment, it executes as part of the LabWindows/CVI process. When your program terminates, the operating system does not release resources as it does when a process terminates. LabWindows/CVI releases as many resources as it can, but your compiled module might have to do more. Also, if the program suspends for debugging purposes, your compiled module might have to disable interrupts.

When you run a standalone executable, it always executes as a separate process. Thus, the run state change callback facility is not necessary and does not work. External compilers report link errors when you define `__RunStateChangeCallback` in more than one object file. If you require a run state change callback in a compiled module that you intend to use both in LabWindows/CVI and an external compiler, National Instruments recommends that you put the callback function in a separate source file and create a library (`.a`) instead of an object file.

Using Externally Compiled Modules

In general, you can load objects compiled with the Sun compilers and the GNU `gcc` compiler into LabWindows/CVI, with a few restrictions.

Restrictions on Externally Compiled Modules

You can use externally compiled modules with the following restrictions:

- The objects must not use any data types that are incompatible with the LabWindows/CVI compiler or libraries. Incompatible data types include the following:
  - `long double` with any Sun compilers. A Sun compiler implements `long double` as a 16-byte object, but LabWindows/CVI implements it as an 8-byte object.
  - `long long` with the Solaris 2 Sun compiler. LabWindows/CVI does not support this non-ANSI type.
  - Any enumeration type. Many compilers implement enumeration types with different sizes and values.
- You cannot load a Solaris 2 object file when you run LabWindows/CVI under Solaris 1. However, you can load Solaris 1 objects when you run under Solaris 2.

Compiling Modules With External Compilers

You can compile external modules using LabWindows/CVI header files instead of the headers the compiler supplies. To compile this way, you must define the preprocessor macro `_NI_sparc_` to the value `1` for Solaris 1 or to the value `2` for Solaris 2.

When using the Sun ANSI C compiler, use the `-I` flag to add the LabWindows/CVI include directory to the search list, as shown in the following command lines:

```
Solaris 1: acc -Xc -I/home/cvi/include -D_NI_sparc_=1 -c mysource.c
Solaris 2: cc -Xc -I/home/cvi/include -D_NI_sparc_=2 -c mysource.c
```

When using the GNU compiler, use the `-nostdinc` flag to disable the standard include files and the `-I` flag to add the LabWindows/CVI include directory to the search list. Also, you must use the `-ansi` flag. For example, to compile the file `mysource.c` using LabWindows/CVI headers under Solaris 1, use the following command line.

```
gcc -ansi -nostdinc -I/home/cvi/include -D_NI_sparc_=1 -c mysource.c
```

You might see warnings about conflicting types for the built-in functions `memcmp` and `memcpy`, but you can ignore them.

Note ***These examples assume that*** `/home/cvi/include` ***is the LabWindows/CVI header files directory. The actual path depends on how you install your copy of LabWindows/CVI.***

You cannot use the non-ANSI C Sun compiler `cc` because it does not recognize some ANSI C constructs in the header files, such as function prototypes and the keywords `const`, `void`, and `volatile`.

Locking Process Segments into Memory Using plock()

You can use the UNIX function `plock` to lock the text and data segments of your program into memory. However, this function locks all segments of the LabWindows/CVI process, not just the segments associated with your program. Also, because the text segments of LabWindows/CVI programs actually reside in the data segment of the LabWindows/CVI process, you must lock both text and data segments, using `plock(PROCLOCK)`, in order to lock all text into memory.

☞ **Note** ***Your LabWindows/CVI process must have superuser privileges to use the*** `plock` ***function.***

UNIX Asynchronous Signal Handling

The following signals have special meaning in LabWindows/CVI:

- **SIGPOLL (SIGIO) and SIGPIPE**—The LabWindows/CVI TCP Library installs signal handlers for `SIGPOLL` (`SIGIO`) and `SIGPIPE`. If you use the TCP Library and you want to install handlers for these signals, you must call the LabWindows/CVI handlers when your handlers are called. If you attempt to set the signal handler to `SIG_DFL` for these signals while running in the LabWindows/CVI environment, LabWindows/CVI restores its own handlers.
- **SIGINT and SIGQUIT**—Normally, the operating system generates these two signals when you type certain keystrokes (<Ctrl-C> and <Ctrl-\>) in the window from which you invoke LabWindows/CVI. If one of these signals occurs while your program is running and you have not installed a handler for it, LabWindows/CVI suspends your program the next time it calls a function that processes events (such as `ProcessSystemEvents`). If your program does not call any event-processing functions, it continues to run.
- **SIGTERM**—LabWindows/CVI treats `SIGTERM` as a stronger version of `SIGINT` and `SIGQUIT`. If this signal occurs while your program is running and you have not installed a handler for it, LabWindows/CVI terminates the program, gives you a chance to save your files, and exits. If `SIGTERM` occurs when no program is running, LabWindows/CVI exits immediately.
- **SIGBUS, SIGFPE, SIGILL, and SIGSEGV**—These signals exist to allow for hardware exceptions. Because execution cannot continue beyond the instruction that caused the exception, LabWindows/CVI always catches these signals. If this signal occurs while your program is running, LabWindows/CVI reports a fatal run-time error

and suspends operation at the statement that caused the exception. If this signal occurs when no program is running, LabWindows/CVI exits immediately.

You cannot use `signal`, `sigaction`, `sigset`, or `sigvec` to make your program ignore the signals this section lists.

☞ **Note** ***If your program begins to loop indefinitely, you can often suspend execution by sending a signal to the LabWindows/CVI process as follows:***

1. ***Use the*** `ps` ***command to identify the process number of LabWindows/CVI.***
2. ***Send the*** `kill -SIGNAL pid` ***command to that process. For example, if the LabWindows/CVI process number is 3478, the command*** `kill -INT 3478` ***sends the*** `SIGINT` ***signal to LabWindows/CVI. When you want to suspend execution of your program in LabWindows/CVI, try using*** `SIGINT` ***or*** `SIGQUIT`***. If sending the*** `SIGINT` ***or*** `SIGQUIT` ***signal fails, you must use the stronger*** `SIGTERM` ***signal, which terminates not just your program but also LabWindows/CVI.***

☞ **Note** ***Some signals can cause LabWindows/CVI to dump core you are running a program that does not install handlers for them.***

Solaris 1 ANSI C Library Implementation

The C library that comes with Sun Solaris 1 (SunOS 4.1.*x*) does not comply with the ANSI C standard as follows:

- Some ANSI C functions are missing from the library.
- Some library functions have different behavior than the ANSI standard specifies.

LabWindows/CVI corrects these problems by linking your programs to a supplemental C library `libcfix.a`, which is in the `lib` subdirectory of the LabWindows/CVI installation directory. This library contains replacement functions for some Sun Solaris functions and for the ANSI functions that are not available in the Sun Solaris library. The names of the replacement functions differ from the Sun Solaris function names and do not interfere with programs or libraries that depend upon the non-ANSI behavior of some Sun Solaris functions. The LabWindows/CVI ANSI header files contain macro definitions for the replacement functions. When you compile with the LabWindows/CVI headers, your program references the LabWindows/CVI replacement functions instead of the Sun Solaris versions.

Consider the case of `realloc`, which LabWindows/CVI replaces with `_cvi_realloc`. The Sun Solaris 1 implementation of the `realloc` function fails when the first argument is NULL. The ANSI standard requires that `realloc` accept NULL as a first argument. In the library `libcfix.a`, LabWindows/CVI defines `_cvi_realloc`, which treats a NULL argument as the ANSI standard prescribes. The LabWindows/CVI header file `stdlib.h`

contains the following macro definition so that `_cvi_realloc` replaces all references to `realloc` in your program.

```
#define realloc _cvi_realloc
```

Note ***Object files you previously compiled using either older LabWindows/CVI headers or Sun Solaris headers do not reference the replacement functions. You must recompile your object files using LabWindows/CVI headers to obtain ANSI-compliant behavior.***

The following lists show the complete contents of the supplemental C library `libcfix.a`.

Replacement Functions

Name	Header	Non-ANSI Behavior of Sun Version
`_cvi_fflush`	`stdio.h`	Does not handle NULL argument properly.
`_cvi_fopen`	`stdio.h`	Does not support binary open mode ("b"). Append open mode ("a") incorrect.
`_cvi_freopen`	`stdio.h`	Same as fopen.
`_cvi_realloc`	`stdlib.h`	Does not handle NULL argument properly.
`_cvi_strtol`	`stdlib.h`	Does not set errno to `ERANGE` on error.
`_cvi_system`	`stdlib.h`	Does not handle NULL argument properly.
`matherr`		Default behavior prints error message.

Additional Functions Not Found in Sun Solaris 1 libc

`_assert` (used by `assert()` macro in `assert.h`)
`labs`
`srand`
`fsetpos`
`fgetpos`
`atexit`
`difftime`
`div`
`ldiv`
`fpos`
`memmove`
`raise`
`rand`
`strerror`
`strtoul`

Incompatibilities among LabWindows/CVI, Sun Solaris, and ANSI C

Under the ANSI C standard, the programmer who implements the library chooses how certain functions behave. As a result, two implementations of a function can behave differently and still conform to the ANSI standard. Because LabWindows/CVI now uses the Sun Solaris C library, incompatibilities arise from the following sources:

- Differences between LabWindows/CVI and the ANSI standard
- Differences between LabWindows/CVI and the Sun Solaris standard

This section outlines these incompatibilities.

Note ***None of these incompatibilities interfere with development of projects and standalone executables in LabWindows/CVI for Sun.***

Between LabWindows/CVI and ANSI C

The following incompatibilities exist between LabWindows/CVI and ANSI C:

- LabWindows/CVI for Solaris 1 defines `size_t` as a signed integer instead of an unsigned integer as the ANSI C standard requires. National Instruments uses the signed integer definition to make LabWindows/CVI compatible with the Sun Solaris header files.
- LabWindows/CVI for Solaris 1 uses the Sun Solaris version of the ANSI function `strftime`, which incorrectly interprets the "`%W`" control string as the one-based week number instead of the zero-based week number as ANSI specifies.
- In LabWindows/CVI for Solaris 1, `ungetc` works improperly in certain cases:

  `-fsetpos` fails to erase all memory of pushback characters by `ungetc`.

  `-ungetc` does not clear the end-of-file indicator on success.

  `-ungetc` fails in certain cases after reading to the end of a file.
- In LabWindows/CVI for Solaris 1, signal handlers you install with the `signal` function remain installed after invocation of the handler. The ANSI standard specifies that these handlers be removed before they are invoked.
- In standalone executables LabWindows/CVI creates for Solaris 1, the function `abort` and the macro `assert` do not terminate the program if you install a signal handler for the `SIGABRT` signal and the handler returns rather than calling `longjmp`. The ANSI standard specifies that the program terminate in these cases.
- Under some versions of Sun Solaris 2 (for example, Solaris 2.5), the ANSI function `setlocale` does not work properly when running programs in the LabWindows/CVI development environment. LabWindows/CVI links programs in the development environment to the Sun Solaris static library `libc.a`, which contains a limited version

of `setlocale`. In contrast, LabWindows/CVI links standalone executables to the shared library `libc.so`, which contains the fully functional version of `setlocale`.

Between LabWindows/CVI and Sun Solaris

The following incompatibilities exist between LabWindows/CVI and Sun Solaris:

- LabWindows/CVI does not support the `long long` data type some header files on Solaris 2 use. In LabWindows/CVI, you cannot use that data type or call functions that use that data type.
- LabWindows/CVI implements the data type `long double` as an 8-byte object, in the same way that it implements `double`. Sun Solaris implements `long double` as a 16-byte object. As a result, Sun Solaris functions that use `long double` do not work properly in LabWindows/CVI.
- The LabWindows/CVI implementation of the `printf` and `scanf` family of functions does not support the Sun Solaris implementation of `long double`.
- LabWindows/CVI does not support wide character constants (`wchar_t`) of the form `L'ab'`.
- The data types `jmp_buf` and `sigjmp_buf` that the header file `setjmp.h` defines are different for LabWindows/CVI and Sun Solaris. The LabWindows/CVI versions of these buffers are larger than the Sun Solaris versions because LabWindows/CVI stores additional debugging information in them. As a result, you must be careful when you use `jmp_buf` and `sigjmp_buf` objects among multiple files. In particular, if you compile a file in LabWindows/CVI with debugging enabled and the file uses `setjmp` or `longjmp`, then your program must include the LabWindows/CVI version of `setjmp.h` to handle those functions correctly. The same is true for `sigjmp_buf`, `sigsetjmp`, and `siglongjmp`.

6

Building Multiplatform Applications

This chapter contains guidelines and caveats for writing platform-independent LabWindows/CVI applications. LabWindows/CVI currently runs under Windows 3.1 and Windows 95/NT for the PC, and Solaris 1 and Solaris 2 for the SPARCstation.

One major feature of LabWindows/CVI is that it supports multiplatform programming. Following a few simple guidelines assures the portability of a LabWindows/CVI application:

- Write code in strict ANSI C.
- Observe and repair all LabWindows/CVI compile, link, and run-time diagnostics.
- Avoid using system dependent calls when possible.
- Avoid using non-portable image formats and fonts in your user interface.

Multiplatform Programming Guidelines

LabWindows/CVI is portable because it uses ANSI C program files, LabWindows/CVI User Interface Resource files, and National Instruments libraries.

You must segregate any platform dependent code in your source code using conditional preprocessor directives. You can use the built-in macros, such as `_NI_mswin32_`, `_NI_mswin16_`, `_NI_mswin_`, `_NI_unix_` and `_NI_sparc_`. More information on the macros that LabWindows/CVI automatically defines is available in the *Compiler Defines* section of Chapter 1, *LabWindows/CVI Compiler*.

Library Issues

Avoid using Windows 32-bit SDK functions unless you intend your LabWindows/CVI application to run only under Windows 95/NT.

The `sopen` and `fdopen` functions are available only under Windows. Avoid using them unless you intend your LabWindows/CVI application to run only under Windows.

Avoid using UNIX host system library calls such as `ioctl`, `fcntl`, and so on, unless you intend the LabWindows/CVI application to run only under UNIX. Refer to the *Using the*

Low-Level I/O Functions section in Chapter 1, *LabWindows/CVI Compiler*, of this manual, for more information on how to use system library calls.

Under UNIX, the low-level I/O functions `open`, `close`, `read`, `write`, `lseek`, and `eof` are available in the UNIX C library. Refer to the *Using the Low-Level I/O Functions* section Chapter 1, *LabWindows/CVI Compiler*, of this manual, for more information on how to use UNIX C low-level functions. These functions are portable to Windows if you include `lowlvlio.h` in your Windows application.

In general, the ANSI C, User Interface, Analysis, Formatting and I/O, Utility, GPIB, VXI, RS-232, and TCP libraries are portable across platforms. However, a few functions are not multi-platform. The majority of these functions are in the Utility Library. The documentation and function panels for the non-portable functions contain notes that list the platforms to which they apply.

Only LabWindows/CVI for Windows has DDE, Data Acquisition, and Easy I/O for DAQ libraries. The X Property Library is available only under UNIX. The ActiveX Automation Library is available only under Windows 95/NT.

Although LabWindows/CVI provides the TCP Library on all platforms, you are responsible for ensuring that the system has hardware and software support for the TCP server.

Various processor architectures store integers and floating point numbers in different byte order. To circumvent these inconsistencies, use the `[o]` modifier in the Formatting and I/O Library to describe the byte ordering of device data. In a `Fmt`/`Scan` function, use the `[o]` modifier to describe the byte ordering for the buffer that contains the raw device data. Do not use the `[o]` modifier on the buffer that holds the data in the byte ordering of the host processor. For example, if you use a GPIB instrument that sends two-byte binary data in Intel byte order, use the following code:

```
short instr_buf[100];
short prog_buf[100];
status = ibrd (ud, instr_buf, 200);
Scan (instr_buf, "%100d[b2o01]>%100d[b2]", prog_buf);
```

If you use a GPIB instrument that sends two-byte binary data in Motorola byte order, use `Scan` as shown in the following example:

```
Scan (instr_buf, "%100d[b2o10]>%100d[b2]", prog_buf);
```

In either case, use the `[o]` modifier only on the buffer that contains the raw data from the instrument (`instr_buf`). LabWindows/CVI ensures that the program buffer (`prog_buf`) uses the proper byte order for the host processor. For a full description of the `[o]` modifier, refer to Chapter 2, *Formatting and I/O Library*, of the *LabWindows/CVI Standard Libraries Reference Manual*.

Externally Compiled Modules

Although you can use externally compiled modules in LabWindows/CVI as this manual describes, the best medium for application portability is ANSI C source code. Object modules are not directly portable from one platform to another because the object file formats on the various platforms differ.

For example, the object file formats are different among Windows 3.1, Windows 95/NT, and UNIX systems. Although SPARCstations have the same computer architecture, Solaris 1.*x* (Sun OS 4.*x*) and Solaris 2.*x* also use different object file formats that make object modules non-portable even between these two systems.

To use an externally compiled module across platforms, you must recompile the source code for the module with a compiler for the target system.

Multiplatform User Interface Guidelines

Function panel (`.fp`) files are portable across platforms.

User Interface Resource (`.uir`) files are portable across platforms.

Image file formats other than PCX (`.pcx`) are not portable.

Color hue and intensity differences between platforms are unavoidable.

The only fonts sure to be available on all platforms are the National Instruments fonts. National Instruments fonts of the same name resemble each other stylistically from one platform to another, although some relative size differences might exist. The National Instruments Meta Fonts are of uniform size (height) relative to the rest of the user interface and are the most portable family of fonts available. However, the width of the National Instruments Meta Fonts might differ slightly from one platform to another. Allow for extra space in the width of all control labels to assure consistent appearance.

You might find the User Interface library functions `GetCtrlBoundingRect`, `GetTextDisplaySize`, and `GetScreenSize` useful in calculating and compensating for font-size discrepancies between platforms.

The order in which LabWindows/CVI processes user interface events might differ between Windows and UNIX platforms. This happens because of differences between the underlying window management systems that LabWindows/CVI uses.

You must not assign the forward <Delete> key as a hot-key in your user interface, because that key does not exist on all platforms.

7

Creating and Distributing Standalone Executables and DLLs

This chapter describes how the LabWindows/CVI Run-time Engine, DLLs, externally compiled modules, and other files interact with your executable file. This chapter also describes how to perform error checking in a standalone executable program. You can create executable programs from any project that runs in the LabWindows/CVI environment.

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 Standalone Executables under Windows

Under Windows, you can bundle the LabWindows/CVI Run-time Engine with your distribution kit using the **Create Distribution Kit** command in the **Build** menu of the Project window, or you can distribute it separately by making copies of the Run-time Engine.

Minimum System Requirements for Windows 95/NT

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

- Windows 95, or Windows NT version 4.0 or later
- A personal computer with at least a 33 MHz 486 or higher microprocessor
- A VGA resolution or higher video adapter
- A minimum of 8 MB of memory
- Free hard disk space equal to 4 MB, plus space to accommodate your executable or DLL and any files the executable or DLL requires

No Math Coprocessor Required for Windows 95/NT

You do not have to have a math coprocessor or emulator to use the LabWindows/CVI Run-time Engine under Windows 95/NT.

Minimum System Requirements for Windows 3.1

To run a standalone executable you create using LabWindows/CVI for Windows, you must have the following:

- MS-DOS, version 3.1 or later
- Microsoft Windows operating system, version 3.1 or later
- A personal computer with at least a 25 MHz 386 or higher microprocessor. National Instruments recommends a 33 MHz 486 or higher microprocessor.
- A VGA resolution or higher video adapter
- A math coprocessor
- A minimum of 4 MB of memory
- Free hard disk space equal to 2 MB, plus space to accommodate your executable and any files the executable requires

Math Coprocessor Software Emulation for Windows 3.1

To run a standalone executable you create using LabWindows/CVI for Windows 3.1, your system must have a math coprocessor. LabWindows/CVI recognizes the following coprocessor emulation programs.

- `wemu387.386` from Watcom
- Q387 from Quickware

Distributing Standalone Executables under UNIX

The **Create Distribution Kit** command is not available with UNIX versions of LabWindows/CVI. However, you can use one of several UNIX shell scripts in the `misc/bin` directory of the LabWindows/CVI installation directory to package your standalone programs for distribution.

Distributing Standalone Executables under Solaris 2

To use the System V software packaging utility `pkgmk` to distribute executable programs under Solaris 2, complete the following steps.

1. If your program loads `.uir` files with `LoadPanel` or loads external modules with `LoadExternalModule`, use caution when you specify the filenames in calls to these functions. If you use a relative path, the path is relative to the directory that contains the executable. Refer to the *Location of Files on the Target Machine for Running Executables and DLLs* section later in this chapter for more information.
2. Create a directory to contain your executable program and associated files. Structure the directory exactly as you want it to appear after installation. Test your program by running it from that directory.
3. From the directory that contains your executable program and associated files, execute the `makepkg` shell script in the `misc/bin` directory of the LabWindows/CVI installation directory to create a distribution package. The script requires the following information to build the package:
   - Abbreviated package name that can have up to nine characters in the form *XYZmyapp*
   - Text name for the package
   - Default installation base directory on the user's machine
   - Directory to place the build package

   The script requests the following information, which is optional:
   - Company or vendor name for the package
   - Name and path to a copyright notice file for the package
   - Relative path and executable name to create as a symbolic link
4. The `makepkg` script creates the following files and directory structure. In the following paths, *pkgname* stands for the name of your application package.

   pkgname`/install/copyright`

   pkgname`/install/postinstall`

   pkgname`/install/preremove`

   pkgname`/pkginfo`

   pkgname`/pkgmap`

   pkgname`/reloc/`*pkgname*`/contents of application directory`

   You can now place the *pkgname* directory and its contents onto your distribution media.
5. To run your executable, you must have the LabWindows/CVI Run-time Engine. You can build the package for the LabWindows/CVI Run-time Engine by executing `makecvirte` located in the `misc/bin` directory of the LabWindows/CVI installation directory. The

makecvirte script prompts you to name the directory in which to place the completed package. The package name is NICcvirte.

6. To install or remove a package on a machine you must log in as root. You can then use either of the following two methods to install or remove a package:
   - Use the Software Management Tool swntool located in the /usr/sbin directory of your system.
   - Use the following command to install a package:

     pkgadd -d *<path to package> pkgname*

     To remove a previously installed package, issue the following command:

     pkgrm *pkgname*

Distributing Standalone Executables under Solaris 1

To distribute executable programs under Solaris 1, complete the following steps.

1. If your program loads UIR files with LoadPanel or loads external modules with LoadExternalModule, use caution when you specify filenames in calls to these functions. If you use a relative path, the path is relative to the directory that contains the executable. Refer to the *Location of Files on the Target Machine for Running Executables and DLLs* section in this chapter for more information.
2. Create a directory containing your executable program and associated files. Structure the directory exactly as you want it to appear after installation. To test your program, run it from that directory.
3. Use the shell script makedist in the misc/bin directory to create a distribution package. This script creates a compressed tar file that contains the directory you created in Step 2 and a copy of the LabWindows/CVI Run-time Engine.
4. Make a copy of the installation script INSTALL.sample in the misc/bin directory and customize it using the information makedist provides. This installation script unpacks a distribution package, creating a directory like the one you created in Step 2, and then installs the LabWindows/CVI Run-time Engine. The installation script can install from floppy disks or from the current directory.
5. If you want to distribute your program on floppy disks, use the shell script makefloppy in the misc/bin directory to copy your installation script and distribution package to floppy disks. If you want to distribute using some other method, such as anonymous FTP, you must provide users with the package file that makedist creates and the customized installation script that extracts the files from the package.

You can use this method under Solaris 2 if you do not want to use the pkgmk utility.

Minimum System Requirements for UNIX

To run a standalone executable you create using LabWindows/CVI for UNIX, your system must have the following:

- Sun SPARCstation
- Solaris 1.*x* (SunOS 4.1.2 or higher) or Solaris 2.4 or higher
- At least 24 MB of RAM
- At least 32 MB of disk swap space
- Free hard disk space equal to 4 MB, plus space to accommodate your executable and any files the executable requires

Translating the Message File

The message file, called `msgrt`*n*`.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. It resides in the `bin` directory of the Run-time Engine installation directory. You can translate the message file into other languages. To translate the message file, perform the following steps.

1. Copy the file to another name so you have it as a backup.
2. Use a text editor to modify `msgrt`*n*`.txt`. Translate only the text that is inside quotation marks. You must 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 in the following example:

   ```
   countmsg msgrt5.txt
   ```

Configuring the Run-Time Engine

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

Solaris 1 Patches Required for Running Standalone Executable

Executables you create using LabWindows/CVI do not run properly on some versions of Solaris 1 (SunOS 4) unless you patch the dynamic linker (`/usr/lib/ld.so`). For this reason, you might have to patch the operating system on the machine on which you install your standalone executable.

The required patches are available from Sun and also come with LabWindows/CVI. You can either install the patch automatically using the installation script in the directory that contains the patch, or you may install the patch manually by following the instructions that come with

the patch. The required patches are available in the `misc/patch` subdirectory of the LabWindows/CVI installation directory. The following patches are available:

- Patch-ID #100257-06 for SunOS 4.1.3/4.1.3c
- Patch-ID #101743-02 for SunOS 4.1.3_U1
- Patch-ID #101783-02 for SunOS 4.1.1/4.1.2

Configuration Option Descriptions

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

Refer to the *How to Set the Configuration Options* discussion in Chapter 1, *Configuring LabWindows/CVI*, of the *LabWindows/CVI User Manual* for detailed instructions on how to manually set configuration options on each platform for the LabWindows/CVI development environment. Under UNIX, you set the Run-time Engine configuration options in the same manner. Under Windows, you set the Run-time Engine configuration options in a similar manner, but with the following differences:

- Under Windows 95/NT, set the configuration options in the Registry under the following key:

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

- Under Windows 3.1, set the configuration options in the [`cvirt`*n*] section of the `win.ini` file, where *n* is the version of the Run-time Engine.

☞ **Note** ***Under UNIX, changes to options do not take effect until you restart your X server or issue the*** `xrdb .Xdefaults` ***command.***

cvirt*x* (Windows 3.1 Only)

Because executables load and execute the Run-time Engine under Windows 3.1, they must be able to locate the Run-time Engine on the hard disk. Under Windows 3.1, executables find the Run-time Engine using `cvirt`*n*, where *n* is the version number of the Run-time Engine, configuration option.

Assign the pathname of the Run-time Engine executable file to the `cvirt`*n* option in the [`cvirt`*n*] section of `win.ini`, as in the following example:

```
[cvirt5]
cvrt5=c:\windows\system\cvirt5\cvirt5.exe
```

cvidir (Windows Only)

Under Windows 95/NT, `cvidir` specifies the location of the directory that contains the `bin` and `fonts` subdirectories that the Run-time Engine requires. This Registry entry is necessary to enable the Windows 95/NT Run-time Engine DLL to load. When you install the LabWindows/CVI Run-time Engine under Windows 95/NT, the installation program places the `bin` and `fonts` subdirectories in the `cvirte` directory under the Windows system directory. The installation program also creates the `cvidir` entry in the Registry.

For Windows 3.1, set the `cvidir` option only if the Run-time Engine resides in a directory other than the directory that contains the `bin` and `fonts` subdirectories. Set it to the directory that contains the `bin` and `fonts` subdirectories.

useDefaultTimer (Windows Only)

The LabWindows/CVI Run-time Engine recognizes the `UseDefaultTimer` option under Windows platforms. It has the same effect as in the LabWindows/CVI development environment. Refer to Chapter 1, *Configuring LabWindows/CVI*, in the *LabWindows/CVI User Manual*, for more information on `useDefaultTimer`.

DSTRules

The LabWindows/CVI Run-time Engine recognizes the `DSTRules` option. It has the same effect as in the LabWindows/CVI development environment. Refer to Chapter 1, *Configuring LabWindows/CVI*, in the *LabWindows/CVI User Manual*, for more information on `DSTRules`.

UNIX Options

The LabWindows/CVI Run-time Engine recognizes the `activate`, `appFont`, `dialogFont`, `editorFont`, `menuFont`, `messageBoxFont`, `useDefaultColors`, `useMetaKey`, and `warpMouseOverDialogBoxes` options under UNIX platforms. They have the same effect as in the LabWindows/CVI development environment. Refer to Chapter 1, *Configuring LabWindows/CVI*, in the *LabWindows/CVI User Manual*, for more information on these options.

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 the necessary files to install your LabWindows/CVI executable program on a target machine as shown in Figure 7-1.

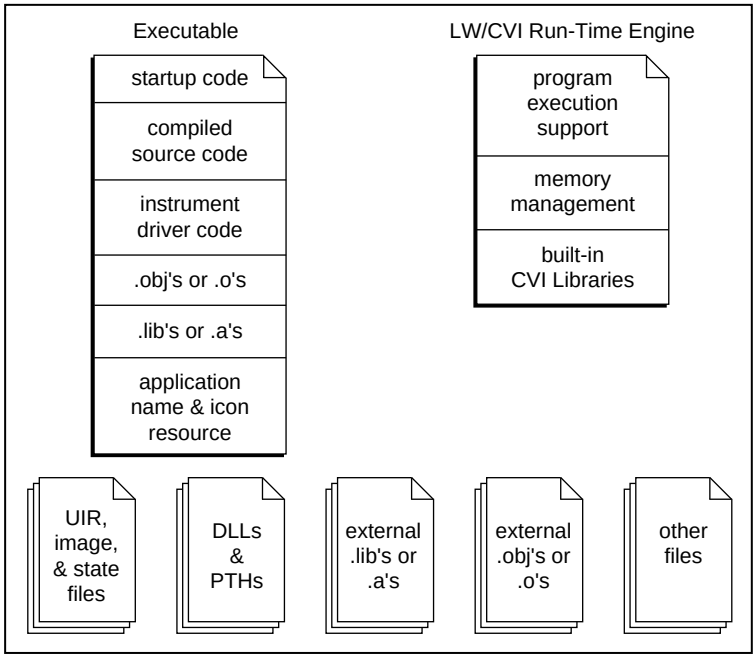

Figure 7-1. Files Necessary to Run a LabWindows/CVI Executable Program on a Target Machine

- **Executable—**This file contains a precompiled, prelinked version of your LabWindows/CVI project and any instrument driver program files that you link to your project. It also contains the application name and icon resource to register to the operating system. The executable has an associated icon on which you can double-click to start the application. When the executable starts, it loads the Run-time Engine. Under UNIX, the executable returns the value that `main` returned or the value you passed to `exit`.
- **Run-time Engine—**The Run-time Engine contains all the built-in library, memory, and program execution code present in the LabWindows/CVI environment, without all the program development tools such as the source editor, compiler, debugger, and user interface editor. The Run-time Engine is smaller than the LabWindows/CVI environment and thus loads faster and requires less memory. You use only one copy of the Run-time Engine on each target machine even when you have multiple executables. Under

Windows 95/NT, the Run-time Engine consists of multiple files, including three DLLs and the low-level support driver. Under Windows 3.1, the Run-time Engine is an execute-only version of the LabWindows/CVI environment. Under Sun Solaris, the Run-time Engine is a shared library

- **UIR files—**The User Interface Resource files that your application program uses. Use `LoadPanel` and `LoadMenuBar` to load these files.
- **Image files—**The graphical image files that you programmatically load and display on your user interface using `DisplayImageFile`.
- **State files**—The user interface panel state files that you save using `SavePanelState` and load using `RecallPanelState`.
- **DLL files**—(Windows Only) the Windows Dynamic Link Library files that your application program uses.
- **PTH files**—(Windows 3.1 Only) specify the location of DLL files when you want to load the DLL from a special directory, or indicate that you want to find a DLL using the standard Windows DLL search algorithm.
- **External .lib or .a files**—Compiled 32-bit `.lib` files on the PC or `.a` files under UNIX that you load using `LoadExternalModule` and that you have not listed in the project.
- **External .obj or .o files**—Compiled 32-bit `.obj` files on the PC or `.o` files under UNIX 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.

Necessary Files for Using DLLs Created in Windows 95/NT

Under Windows 95/NT, you can distribute DLLs that use the LabWindows/CVI Run-time Engine. As in the case of standalone 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. On the PC, you specify these files in a relative directory structure in the dialog box that appears when you select **Create Distribution Kit** from the **Build** menu of the Project window in LabWindows/CVI. Refer to the *LabWindows/CVI User Manual* for details. This section describes the proper location of each of the files shown in Figure 7-1.

LabWindows/CVI Run-Time Engine under Windows 95/NT

Table 7-1 shows the files that comprise the LabWindows/CVI Run-time Engine for Windows 95/NT.

Table 7-1. LabWindows/CVI Run-Time Engine Files

Run-Time Engine File	Description
`cvirt.dll`	Helper DLL
`cvirte.dll`	Contains most LabWindows/CVI libraries
`cviauto.dll`	Contains ActiveX Automation Library
`cvi95vxd.vxd`	Low-level support driver for Windows 95
`cvintdrv.sys`	Low-level support driver for Windows NT
`msgrt`*n*`.txt`	Contains text messages; *n* is Run-time Engine version number
`cvirt`*n*`.rsc`	Contains binary resources; *n* is Run-time Engine version number
`ni7seg.ttf`	Font description file
`nisystem.ttf`	Font description file

These files come on a separate diskette, or in a separate directory in the CD-ROM. The LabWindows/CVI installation program installs the files along with the development environment. The **Create Distribution Kit** command in the **Build** menu of the Project window can bundle the Run-time Engine DLLs and drivers into your distribution kit. Also, you can make copies of this diskette, or the CD-ROM directory, for separate distribution.

Run-Time Library DLLs

The installation program always places the Run-time Engine DLLs in the Windows `system` directory under Windows 95 and the Windows `system32` directory under Windows NT.

Low-Level Support Driver

The Run-time Engine loads the low-level support driver if it is present when you start your standalone executable. Several functions in the Utility Library require the low-level support driver. Refer to the function reference for `CVILowLevelSupportDriverLoaded` in Chapter 8, *Utility Library*, of the *LabWindows/CVI Standard Libraries Reference Manual* for more information on these functions.

The installation program installs the low-level support driver in the Windows `system` directory under Windows 95 and the Windows `system32\drivers` directory under

Windows NT. Under Windows NT, the installation program also adds a registry entry under the following key:

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services\cvintdrv
```

Table 7-2 shows the values the installation program sets for the Windows NT registry entry for the low-level support driver.

Table 7-2. Windows NT Registry Entry Values for the Low-Level Support Driver

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. It installs the `msgrtn.txt` and `cvirt`*n*`.rsc` in the `cvirte\bin` subdirectory under the directory in which it installs the Run-time Engine DLLs. It sets the `cvidir` option in the following registry key to the pathname of the `cvirte` directory.

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

You can subsequently change the location of the `bin` and `fonts` subdirectories, but you must also change 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.

LabWindows/CVI Run-Time Engine under Windows 3.1

For Windows 3.1, the LabWindows/CVI Run-time Engine comes in the form of an executable file. The name of the executable file is `cvirt`*n*`.exe`, where *n* is the version of the Run-time Engine. The Run-time Engine comes on a separate diskette. The LabWindows/CVI installation program installs the Run-time Engine along with the development environment. The **Create Distribution Kit** command in the **Build** menu of the Project window can bundle

the Run-time Engine into your distribution kit. Also, you can make copies of this diskette for separate distribution. The user selects the directory into which to install the Run-time Engine.

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

LabWindows/CVI Run-Time Engine under Sun Solaris

Under Sun Solaris, the Run-time Engine comes in the form of a shared library called `libcvi.so.n`, where *n* is the version number of the Run-time Engine. The installation program installs the shared library in the `lib` subdirectory under the Run-time Engine installation directory. It also creates two symbolic links, where *`cvirte`* stands for the Run-time Engine installation directory, as shown in Table 7-3.

Table 7-3. Pathnames and Targets of Links

Pathname of Link	Target of Link
`/usr/lib/libcvi.so`	`/usr/lib/libcvi.so.`*n*
`/usr/lib/libcvi.so.`*n*	*cvirte*`/lib/libcvi.so.`*n*

The installation program installs message and resource files in the `bin` directory under the Run-time Engine installation directory. It installs font description files in the `fonts` directory under the Run-time Engine installation directory. You can subsequently change the location of the `bin` and `fonts` subdirectories, but you must also change the `cvidir` configuration option to the pathname of the directory that contains the two subdirectories.

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

Rules for Accessing UIR, Image, and Panel State Files on All Platforms

The recommended method for accessing UIR, image, and panel state files in your executable program is to place the files in the same directory as the executable and pass simple filenames with no drive letters or directory names to `LoadPanel`, `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`, `DisplayImageFile`, `SavePanelState`, and `RecallPanelState`. These functions interpret relative pathnames as being relative to the directory that contains the executable.

Rules for Using DLL Files under Windows 95/NT

Under Windows 95/NT, your executable or DLL can link to a DLL only through an import library. This section refers to a DLL 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 it in your project.
- Associate it with the `.fp` file for an instrument driver or user library.
- Dynamically load it by a 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. On the other hand, if you load the import library through a call to `LoadExternalModule`, you must distribute it separately from your executable. Refer to the *Rules for Loading Files Using LoadExternalModule* section later in this chapter for more information.

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 95, the Windows `system` directory. Under Windows NT, 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 Using DLL Files under Windows 3.1

LabWindows/CVI never links DLL files and DLL path files into the executable, so you must distribute them as separate files. The **Create Distribution Kit** command automatically includes DLLs that your project refers to in your distribution kit. The only exceptions are DLLs for National Instruments hardware and DLLs that you load using `LoadExternalModule`.

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.

If you use `LoadExternalModule` to load DLLs, refer to the following section, *Rules for Loading Files Using LoadExternalModule*.

If you use a DLL file, a DLL path file, or a DLL glue object module in your project or as an instrument driver, the Run-time Engine always looks for a corresponding DLL path (`.pth`) file before it looks for the DLL itself. This search mechanism lets the user of your executable place the DLLs anywhere on the target computer. The Run-time Engine uses the following DLL search method.

1. Look for a `.pth` file in the directory of the executable. The `.pth` file must have the same base name as the file in the project or as the instrument driver. If the `.pth` file contains an absolute path to the DLL, use that path to find the DLL. If the `.pth` file contains a simple filename, use the standard Windows DLL search algorithm: directory of executables, current working directory, `\windows`, `\windows\system`, then the `PATH` environment variable.
2. Look for a `.dll` file in the directory of the executable. The `.dll` file must have the same base name as the file in the project or as the instrument driver.
3. Otherwise, use the standard Windows DLL search algorithm.

☞ **Note** ***Before searching for a*** `.dll` ***file, the Run-time Engine always looks for a*** `.pth` ***file. Therefore, your choice of whether to use a*** `.pth` ***file when you develop your application in the LabWindows/CVI environment does not restrict your choice of whether to use a*** `.pth` ***file in the standalone application.***

Rules for Loading Files Using LoadExternalModule

`LoadExternalModule` can load the following file types:

Library Files:	`.lib` (Windows) or `.a` (UNIX)
Object Modules:	`.obj` (Windows) or `.o` (UNIX)
DLL Import Library Files:	`.lib` (Windows 95/NT only)
DLL Path Files:	`.pth` (Windows 3.1 only)
DLL Files:	`.dll` (Windows 3.1 only)
Source Files:	`.c` (linked into your executable or DLL)

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

In the LabWindows/CVI development environment, external modules can link to modules in the **Instrument** and **Library** menus regardless of whether you refer to them elsewhere in your project. However, when you create a standalone executable, 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. You can do this when you create your executable or DLL by using the **Add Files To Executable** or **Add Files To DLL** button to display a list of project `.lib`, project `.a`, Instrument, and Library files. Select the files you want to include in your executable or DLL. If you select a `.lib` or `.a` 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 `.dll` or `.pth` files (Windows 3.1 only), refer to the *Rules for Using DLL Files under Windows 3.1* section earlier in this chapter.
  - For `.lib` files, by default, **Create Standalone Executable File** or **Create Dynamic Link Library** only links in the library modules that you reference statically in the project. 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, use the **Add Files to Executable** button in the Create Standalone Executable File dialog box, or the **Add Files to DLL** button in the Create Dynamic Link Library dialog box.

    - 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.

- If you *link in an external compiler* under Windows 95/NT, 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. Your alternatives are as follows.
    1. Remove the file from the project and distribute it as a separate `.obj`, `.lib`, or `.dll`.
    2. Use the Other Symbols section of the External Compiler Support dialog box in the **Build** menu of the Project 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 it without having to replace the executable. For this reason, if you call `LoadExternalModule` on a library or object in the project, remove or exclude the file from the project before you select **Create Standalone Executable File** or **Create 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 under Windows 95/NT

Under Windows 95/NT, 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. For import libraries that you link into your executable or DLL, refer to the *Using LoadExternalModule on Files in the Project* section earlier in this chapter. For import libraries that you load dynamically, refer to the *Using LoadExternalModule on Library and Object Files Not in the Project* section earlier in this chapter.

You must always 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 application loads
2. The current working directory
3. Under Windows 95, the Windows `system` directory. Under Windows NT, the Windows `system32` and `system` directories
4. The Windows directory
5. The directories the `PATH` environment variable lists

Using LoadExternalModule on DLL and Path Files under Windows 3.1

DLL files and DLL path files are never linked into the executable, so you must distribute them as separate files.

Your executable can call `LoadExternalModule` directly on a DLL or DLL path file only if you include the DLL or DLL path file in the project. When you select **Create Standalone Executable File**, LabWindows/CVI automatically creates the DLL glue code and links it into the executable.

Also, you can pass the DLL glue object module filename to `LoadExternalModule`. You can generate the DLL glue object module by opening the `.h` file for the DLL in a Source window of LabWindows/CVI and selecting **Generate DLL Glue Object** from the **Options** menu.

If you include the DLL, the DLL path file, or the DLL glue object module as a file in the project, you must pass `LoadExternalModule` a simple filename, and it uses the following search method to find the DLL.

1. Look for a `.pth` file in the directory of the executable. The `.pth` file must have the same base name as the file you passed to `LoadExternalModule`. If the `.pth` file contains an absolute path to the DLL, use that path to find the DLL. If the `.pth` file contains a simple filename, use the standard Windows DLL search algorithm: directory of executables,

current working directory, \windows, \windows\system, then the PATH environment variable.

2. Look for a .dll file in the directory of the executable. The .dll file must have the same base name as the file you passed to LoadExternalModule.
3. Otherwise, use the standard Windows DLL search algorithm.

If you maintain the DLL glue object module as a separate file from the executable, you must pass LoadExternalModule a pathname to the DLL glue object module, and it uses the following search method to find the DLL.

1. Look for a .pth file that is in the same directory as the DLL glue object module and that has the same base name as the DLL glue object module. If the .pth file contains an absolute path to the DLL, use it to find the DLL. If the .pth file contains a simple filename, use the standard Windows DLL search algorithm.
2. Look for a .dll file that is in the same directory as the DLL glue object module and that has the same base name as the DLL glue object module.
3. Otherwise, use the standard Windows DLL search algorithm.

Note ***Before searching for a*** .dll ***file, a standalone executable always looks for a*** .pth ***file. Therefore, your choice of whether to use a*** .pth ***file when you develop your application in the LabWindows/CVI environment does not restrict your choice of whether to use to*** .pth ***file in the standalone application.***

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 Standalone Executable File** or **Create 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 earlier in this chapter.

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 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. Under Windows, 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 Properties dialog box for the executable, then it is the initial current working directory. Under UNIX, the initial current working directory is the directory from which you invoke the executable. You can create an absolute path for a file in the executable directory by using `GetProjectDir` and `MakePathname`.

Error Checking in Your Standalone 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 standalone executable program or DLL, all your source modules are compiled. Compiled modules always disable debugging and the Break on Library Errors option, resulting in smaller and faster code. Thus, you must perform your own error checking when you create a standalone executable program or DLL. Refer to Chapter 9, *Checking for Errors in LabWindows/CVI*, for details about performing error checking in your code.

8

Distributing Libraries and Function Panels

This chapter describes how to distribute libraries, add libraries to a user's **Library** menu, and specify library dependencies.

How to Distribute Libraries

You can distribute libraries for other users to include in their **Library** menu. 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** menu. This is useful when the library supports other libraries and contains no user-callable functions.

Adding Libraries to User's Library Menu

Normally, users must manually add libraries to the **Library** menu using the **Library Options** command in the Project window **Options** menu. However, you can insert your libraries into the user's **Library** menu by modifying the user's `cvi.ini` file under Windows 3.1, `.cvi.ini` under UNIX, or the Registry under Windows 95/NT.

Under Windows 3.1 and UNIX, the `modini` program is in the LabWindows/CVI `bin` subdirectory for this purpose. A documentation file called `modini.doc` and the source code are in the same directory.

Under Windows 95/NT, 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** menu under Windows 3.1 and UNIX, your `modini` command file must be:

```
add Libraries LibraryFPFile "c:\newlib\lib1.fp"
add Libraries LibraryFPFile "c:\newlib\lib2.fp"
```

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

To add the libraries to the user's **Library** menu under Windows 95/NT, your `modreg` command file must be:

```
setkey   [HKEY_CURRENT_USER\Software\National Instruments]
appendkey  CVI\@latestVersion
add Libraries LibraryFPFile "c:\newlib\lib1.fp"
add Libraries 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 ***LabWindows/CVI must not be running when you use the*** `modini` ***or*** `modreg` ***program to modify*** `cvi.ini` ***or the Registry. If LabWindows/CVI is running while you use these programs, you will lose your changes.***

Specifying Library Dependencies

When one library you distribute is dependent upon 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 **.FP Auto-Load List** command in the **Edit** menu of the Function Tree Editor window of the dependent library to list the `.fp` files of the libraries upon which it depends. Refer to the *Function Tree Editor* chapter of the *LabWindows/CVI Instrument Driver Developers Guide* for details on this command.

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 Directories** command in the Project window **Options** menu. Also, you can add to the Instrument Directories list by editing `cvi.ini` under Windows 3.1, `.cvi.ini` under UNIX, or the Registry under Windows 95/NT.

National Instruments recommends that your installation program modify `cvi.ini`, `.cvi.ini`, or the Registry, automatically. Under Windows 3.1, the `modini` program is in the LabWindows/CVI `bin` subdirectory for this purpose. A documentation file called `modini.doc` and the source code are in the same directory.

Under Windows 95/NT, 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 under Windows 3.1 and UNIX, your `modini` command file must be,

```
add InstrumentDirectories InstrDir "c:\genlib"
```

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

For LabWindows/CVI to be able to find the dependent file under Windows 95/NT, your `modreg` command file must be,

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

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

Caution ***LabWindows/CVI must not be running when you use the*** `modini` ***or*** `modreg` ***program to modify*** `cvi.ini` ***or the Registry. If LabWindows/CVI is running while you use these programs, you will lose your changes.***

9

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 standalone executables, debugging and the Break on Library Errors 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 standalone executables even if the program works inside the 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.

One method of error checking is to check 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 it to. This chapter describes how LabWindows/CVI reports these status codes and some techniques for checking them.

☞ **Note** ***Status codes are integer values. These values are either common to an entire library of functions, or specific to one function. Libraries that have a common set of codes have a listing at the end of the chapter or manual they appear in. You can find the error message for each integer value there. In addition, 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 function description in the LabWindows/CVI manual set. The error description also appears in the online help of the library function panels in LabWindows/CVI.***

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. A common technique for error checking is to 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 above 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. A section describing how each LabWindows/CVI library reports errors follows this section.

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.

User Interface Library

The User Interface Library routines return a negative value when they detect an error. Some functions, such as `LoadPanel`, return positive values for a successful completion. This library uses a common set of error codes, which the *LabWindows/CVI User Interface Reference Manual* and the function panel help list. You can use the function `GetUILErrorString` to get the error message associated with each User Interface Library error code.

Analysis/Advanced Analysis Libraries

The Analysis and Advanced Analysis Library functions return a negative value when they detect an error. This library uses a common set of error codes, which the *LabWindows/CVI Standard Libraries Reference Manual*, the *LabWindows/CVI Advanced Analysis Reference Manual*, and the function panel help list. You can use the function `GetAnalysisErrorString` to get the error message associated with each Analysis Library error code.

Easy I/O for DAQ Library

The Easy I/O for DAQ Library functions return a negative value when they detect an error. They return a positive value as a warning when they are able to complete their task 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. Refer to the *LabWindows/CVI Standard Libraries Reference Manual* or the function panel help for a listing of the error codes and information on the individual functions. You can use `GetDAQErrorString` to get the error message associated with each Easy I/O for DAQ Library error or warning code.

Data Acquisition Library

The Data Acquisition Library functions return a negative value when they detect an error. They return a positive value as a warning when they are able to complete their task 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.

Refer to the back of the *NI-DAQ Function Reference Manual for PC Compatibles* or the function panel help for a listing of the error codes. You can use `GetNIDAQErrorString` to get the error message associated with each Data Acquisition Library error or warning code.

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 specific VXI function reference manual or the on-line help for a listing of the error codes.

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 Library functions return the same value that they assign to*** `ibsta`***. You can choose to use either 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 your *NI-488.2 Function Reference* and user manuals for detailed information on GPIB global variables and listings of status and error codes. LabWindows/CVI function panel help also has listings of status and error codes.

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 on the result of the function that can also be used to detect a problem. Each function should be checked individually. Refer to the RS-232 section in the *LabWindows/CVI Standard Libraries Reference Manual* or the function panel help for a listing of the error codes and information on the individual functions. 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.***

VISA Library

The VISA Library functions return a negative value when they detect an error. They return a positive value as a warning when they can complete their task but not in the way you might expect. 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 Manual* or the function panel help for a listing 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.

IVI Library

The IVI Library functions return a negative value when they detect an error. This library uses a common set of error codes. Refer to the *LabWindows/CVI Instrument Driver Developers Guide* or the function panel help for a listing 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.

TCP Library

The TCP Library functions return a negative value when they detect an error. This library uses a common set of error codes, which the *LabWindows/CVI Standard Libraries Reference Manual* and the LabWindows/CVI function panel help list. You can use `GetTCPErrorString` to get the error message associated with each TCP Library error code.

DDE Library

The DDE Library functions return a negative value when they detect an error. This library uses a common set of error codes, which the *LabWindows/CVI Standard Libraries Reference Manual* and the LabWindows/CVI function panel help list. You can use the function `GetDDEErrorString` to get the error message associated with each DDE Library error code.

ActiveX Automation Library

The ActiveX Automation Library functions return a negative value when they detect an error. This library uses a common set of error codes. Refer to the *LabWindows/CVI Standard Libraries Reference Manual* or the function panel help for a listing of the error codes and information on the individual functions. You can use `CA_GetAutomationErrorString` to get the error message associated with each ActiveX Automation Library error code.

X Property Library

The X Property Library functions return a negative value when they detect an error. This library uses a common set of error codes, which the *LabWindows/CVI Standard Libraries Reference Manual* and the LabWindows/CVI function panel help list. You can use the function `GetXPropErrorString` to get the error message associated with each X Property Library error code.

Formatting and I/O Library

This library contains the file I/O, string manipulation, and data formatting functions. All functions return negative error codes when they detect an error. 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 it 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 on how the last scanning and formatting function executed. The `GetFmtIOError` function returns a code that contains specific error information on the last

Formatting and I/O Library function that performed file I/O. The `GetFmtIOErrorString` function converts this code into an error string. Refer to the *LabWindows/CVI Standard Library Reference Manual* for more information.

Utility Library

Utility Library functions report error codes as return values. You can check each individual function description in the *LabWindows/CVI Standard Libraries Reference Manual* or in the LabWindows/CVI function panel help to determine the error conditions that can occur in each function.

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, you can consult a publication such as *C: A Reference Manual* cited in the *Related Documentation* section of *About This Manual*. Also, you can use the LabWindows/CVI function panel help to determine the error conditions that can occur in each function.

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 so 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

Errors and Warnings

This appendix contains an alphabetized list of compiler warnings, compiler errors, link errors, DLL loading errors, and external module loading errors generated by LabWindows/CVI.

Table A-1. Error Messages

Error Message	Type Error	Comment
`# flag is valid only with o, x, e, f, and g specifiers.`	Non-Fatal Run-time Error	Ensure that you use the correct format specifier, and that no extra characters exist before the format specifier.
`#elif missing constant expression.`	Compile Error	Ensure that a conditional expression follows `#elif` on the same line.
`#if missing constant expression.`	Compile Error	Ensure that a conditional expression follows `#if` on the same line.
`#ifdef expects an identifier.`	Compile Error	Ensure that an identifier follows `#ifdef` on the same line.
`#ifndef expects an identifier.`	Compile Error	Preprocessor conditional directive `#ifndef` requires an identifier following it on the same line. Make sure that an identifier follows `#ifndef` on the same line.
`#line directive cannot specify line 0.`	Compile Error	`#line` preprocessor directive requires a non-zero line number value.
`#line directive cannot specify line greater than 32767.`	Compile Error	`#line` preprocessor directive cannot set the line greater than 32,767.
`#line directive expects numeric argument.`	Compile Error	`#line` preprocessor directive requires a line number value to be specified following `#line`.
`## at beginning of macro definition.`	Compile Error	## preprocessing token is at the beginning of a macro definition. Ensure that a preprocessing token precedes ##.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`## at end of macro definition.`	Compile Error	## preprocessing token is at the end of a macro definition. Ensure that a preprocessing token(s) follows ##.
`, or ) expected.`	Compile Error	Ensure that the function macro argument list terminates with a) and that a , separates all the macro arguments.
`0 flag is not valid with c, s, p, and n modifiers.`	Non-Fatal Run-time Error	Use of an incorrect format specifier or use of a field width starting with 0 might cause this error.
`Aborted load of library FILE.`	Link Error	Library load operation aborted. A more specific diagnostic of the library load error precedes this message.
`Aborted load of member NAME from library FILE.`	Link Error	Library member load operation aborted. A more specific diagnostic of the library member load error precedes this message.
`Aborted load of object module FILE.`	Link Error	Object file load aborted. A more specific diagnostic of the object file load error precedes this message.
`Absolute segments not supported: segment name NAME.`	PC/Windows Load Error	OMF object file contains a segment to load at an absolute address.
`Anonymous enum declared inside parameter list has scope only for this declaration.`	Compile Warning	Enumeration declared in the parameter list has scope only within the parameter list. As a result, its type is incompatible with all other types. You must declare the enumeration type before declaring function types that use it.
`Anonymous struct declared inside parameter list has scope only for this declaration.`	Compile Warning	Structure declared in the parameter list has scope only within the parameter list. As a result, its type is incompatible with all other types. You must declare the structure type before declaring function types that use it.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Anonymous union declared inside parameter list has scope only for this declaration.`	Compile Warning	Union declared in the parameter list has scope only within the parameter list. As a result, its type is incompatible with all other types. You must declare the union type before declaring function types that use it.
`Argument 4 must be 0 or 1.`	Fatal Run-time Error	Value of the argument to the library function must be `0` or `1`.
`Argument NUMBER must be 0, 1, or 2.`	Fatal Run-time Error	Value of the argument to the library function must be `0`, `1`, or `2`.
`Argument must be a function pointer to the correct type of callback function.`	Non-Fatal Run-time Error	Argument to the function is not a pointer to the expected type of callback function.
`Argument must be an open stream.`	Fatal Run-time Error	Argument to the I/O library function must be one of the standard streams (`stdin`, `stdout`, `stderr`) or a stream you open with the `fopen()` or `freopen()` functions.
`Argument must be character.`	Fatal Run-time Error	Value of the argument to the library function must be less than `256`.
`Array argument too small.`	Fatal Run-time Error	Library function requires an array that is larger than the specified argument. Make sure you declare or allocate the array with sufficient elements for the function call.
`Array argument too small (NUMBER bytes). Argument must contain at least NUMBER bytes (NUMBER elements).`	Fatal Run-time Error	Library function requires an array that is larger than the specified argument. Make sure you declare or allocate the array with the number of elements this error message reports.
`Array index (NUMBER) too large (maximum: NUMBER).`	Non-Fatal Run-time Error	You indexed an array past the last element.
`Assertion error: EXPRESSION.`	Fatal Run-time Error	Value of the argument *`EXPRESSION`* to the Standard C Library macro `assert` is `0`.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Assignment between TYPE and TYPE is compiler-dependent.`	Compile Warning	Although allowed, use caution because an assignment of an `integer` type expression value to an `enum` type target might not correspond to any known enumeration constant for that `enum` type. Depending on the enumeration, the size of the `enum` type can be 1, 2, or 4 bytes and therefore may be incapable of representing all integer values.
`Assignment of invalid pointer value.`	Non-Fatal Run-time Error	Value you assigned to a pointer is an invalid pointer value. Check the right side of the assignment to determine if it is the result of a previous invalid pointer operation.
`Assignment of out-of-bounds pointer: NUMBER bytes before start of array.`	Non-Fatal Run-time Error	Value you assigned to the pointer refers to an invalid location, which is *`NUMBER`* bytes before an array. The right side of the assignment is probably the result of previous illegal pointer arithmetic.
`Assignment of out-of-bounds pointer: NUMBER bytes past end of array.`	Non-Fatal Run-time Error	Value you assigned to the pointer refers to an invalid location, which is *`NUMBER`* bytes past the end of an array. The right side of the assignment is probably the result of previous illegal pointer arithmetic.
`Assignment of pointer to freed memory.`	Non-Fatal Run-time Error	Value you assigned to the pointer is invalid because it refers to a location in dynamic memory that the `free` function deallocated. After memory is free, all pointers into that block of memory are invalid.
`Assignment of uninitialized pointer value.`	Non-Fatal Run-time Error	Value you assigned to the pointer is invalid because it was not initialized. The right side of the assignment is probably an uninitialized local variable or an object in dynamic memory that you allocated with `malloc`. Initialize local variables and dynamic memory before you use them. `calloc` both allocates and initializes dynamic memory.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Assignment to const identifier NAME.`	Compile Error	`const` variables or parameters are read-only values that you cannot modify once initialized. Ensure that no assignment operations modify the identifier.
`Assignment to const location.`	Compile Error	`const` variables or parameters are read-only values that you cannot modify once initialized. Ensure that no assignment operations modify the `lvalue` (such as an array reference, or a pointer dereference) that specifies the `const` location.
`Attempt to free invalid pointer expression.`	Fatal Run-time Error	Pointer value you passed to the `free` function is invalid. It is probably the result of a previous invalid pointer operation.
`Attempt to free pointer to freed memory.`	Fatal Run-time Error	Pointer value you passed to the `free` function refers to a location in dynamic memory that you already deallocated.
`Attempt to free uninitialized pointer.`	Fatal Run-time Error	Pointer value you passed to the `free` function is invalid because you did not initialize it. It is probably an uninitialized local variable. Initialize local variables before you use them.
`Attempt to read beyond end of array.`	Non-Fatal Run-time Error	Source array is not large enough to satisfy the destination specifiers.
`Attempt to read beyond end of string.`	Non-Fatal Run-time Error	Source string is not large enough to satisfy the destination specifiers.
`Attempt to realloc invalid pointer expression.`	Fatal Run-time Error	Pointer value you passed to the `realloc` function is invalid. It is probably the result of a previous invalid pointer operation.
`Attempt to realloc pointer to freed memory.`	Fatal Run-time Error	Pointer value you passed to the `realloc` function refers to a location in dynamic memory that you already deallocated.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Attempt to realloc uninitialized pointer.`	Fatal Run-time Error	Pointer value you passed to the `realloc` function is invalid because you did not initialize it. It is probably an uninitialized local variable. You must initialize local variables before you use them.
`Attempt to write beyond end of array.`	Non-Fatal Run-time Error	Output array is smaller than the given format specifiers and input parameters require.
`Attempt to write beyond end of string.`	Non-Fatal Run-time Error	Output string is smaller than the given format specifiers and input parameters require.
`b modifier must precede o modifier.`	Non-Fatal Run-time Error	If both the `b` and `o` modifiers are present, the `b` modifier must precede the `o` modifier.
`Bad BSS section encountered while reading external module: FILE.`	Object Load Error	Object module is corrupted or is of a type that you cannot load into LabWindows/CVI.
`Bad COFF Library header.`	Object Load Error	Library file you are loading is either corrupted or not in the COFF format.
`Bad COFF Library member header.`	Object Load Error	COFF library you are loading contains a module that is corrupted or in an invalid format.
`Bad location code: OMF record position NUMBER: OMF record type NAME.`	Link Error	Object module is corrupted or is of a type that you cannot load into LabWindows/CVI.
`Bad magic number encountered while reading external module: FILE.`	Link Error	Object module is corrupted or is of a type that you cannot load into LabWindows/CVI.
`Bad method: OMF record position NUMBER: OMF record type NAME.`	Link Error	Object module is corrupted or is of a type that you cannot load into LabWindows/CVI.
`Bad name: OMF record position NUMBER: OMF record type NAME.`	Link Error	Object module is corrupted or is of a type that you cannot load into LabWindows/CVI.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Bad OMF record at position NUMBER: OMF record type NAME.`	PC/Windows Load Error	OMF object file contains an unknown object record. Make sure that the object file is OMF and conforms to the 32-bit format LabWindows/CVI supports.
`Bad relocation record encountered while reading external module: FILE.`	Link Error	Object module is corrupted or is of a type that you cannot load into LabWindows/CVI.
`Bad OMF record at position NUMBER: OMF record type NAME.`	PC/Windows Load Error	OMF object file contains an unknown object record. Make sure that the object file is OMF and conforms to the 32-bit format LabWindows/CVI supports.
`Byte ordering is invalid.`	Non-Fatal Run-time Error	Byte ordering that the `o` modifier specifies is not valid for the size of the integer. The number of digits following the `o` must match the size of the integer, and the digits must fall in the range zero to *size of the integer*–1.
`c modifier valid only with l format specifier.`	Non-Fatal Run-time Error	`c` modifier is only valid for the `l` format specifier.
`The callback function, NAME, specified in the UIR file, does not have required prototype.`	Non-Fatal Run-time Error	You specified the *NAME* function as a callback function for an item in a user interface resource file, but it does not have the correct type to be a callback function. Callback functions must have one of the callback types specified in the user interface library header. The function will not be called.
`The callback function, NAME, specified in the UIR file, is not a known function.`	Non-Fatal Run-time Error	You specified the *NAME* function as a callback function for an item in a user interface resource file, but the function does not exist.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Calling conventions have no effect on variables; calling convention ignored. The position of the calling convention modifier may be incorrect.`	Compile Warning	You placed a calling convention keyword before a variable name. For function pointers, you must place the calling convention to the left of the `"*"`, for example: `int (__cdecl * funptr)();`
`Cannot concatenate wide and regular string literals.`	Compile Warning	Make sure the string literals you concatenate are either both wide string literals or regular string literals.
`Cannot free: memory not allocated by malloc() or calloc().`	Fatal Run-time Error	Pointer value you passed to the function `free` is invalid because it does not point to dynamic memory allocated by `malloc` or `calloc`. `free` can deallocate only pointers you obtain from one of these two functions.
`Cannot generate glue for a function without a prototype: NAME.`	Glue Code Generation Error	In order to generate glue code for a DLL function, you must specify a complete prototype for the function. You must specify the types of the parameters in the prototype.
`Cannot generate glue for a static function: FUNCTION.`	Glue Code Generation Error	You cannot export static functions in a DLL; so it is useless to generate glue code for them.
`Cannot generate glue for a variable argument function: FUNCTION.`	Glue Code Generation Error	In LabWindows/CVI for Windows 3.1, you cannot use DLL functions that accept a variable number of arguments.
`Cannot initialize undefined TYPE.`	Compile Error	You attempted to initialize a declaration of an incomplete `struct` or `union` type, such as a `struct` or `union` type whose members you have not yet specified. Ensure that the initialization appears after the full struct or union declaration.
`Cannot link variable NAME to import library without __import keyword in declaration.`	Link Error	Variable that you have declared as `extern` is defined in a DLL import library, but you did not include the `__import` qualifier in the declaration.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Cannot link variable NAME to import library without declspec(dllimport) keyword in declaration.`	Link Error	Variable that you have declared as `extern` is defined in a DLL import library, but you did not include the `declspec(dllimport)` qualifier in the declaration.
`Case label must be a constant integer expression.`	Compile Error	Case labels must be known integer values at compile time; make sure the case label conforms to the requirements for a constant integer expression.
`Cast from TYPE to TYPE is illegal in constant expressions.`	Compile Error	You cannot cast a pointer type to arithmetic type in a constant expression.
`Cast from TYPE to TYPE is illegal.`	Compiler Error	ANSI C does not allow a cast between the two types.
`COFF Name too long.`	Object Load Error	COFF object or library you are loading contains a symbol name that is longer than the maximum legal length.
`Comparison involving null pointer.`	Non-Fatal Run-time Error	One of the pointer expressions in the comparison has the value `NULL`. Both expressions in pointer comparisons must point into the same array object.
`Comparison involving uninitialized pointer.`	Non-Fatal Run-time Error	One of the pointer expressions in the comparison is invalid because you did not initialize it.
`Comparison of pointers to different objects.`	Non-Fatal Run-time Error	Pointer expressions in the comparison point to two distinct objects. Both expressions in pointer comparisons must point into the same array object.
`Comparison of pointers to freed memory.`	Non-Fatal Run-time Error	One of the pointer expressions in the comparison is invalid because it refers to a location in dynamic memory that you deallocated with the `free` function. Once you free the memory, all pointers into that block of memory become invalid.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Compound statements nested too deeply.`	Compile Error	Program has exceeded the compiler limitations on the number of nested compound statements; reduce the depth of the nested compound statements in the program.
`Conditional inclusion nested too deeply.`	Compile Error	Program has exceeded the compiler limitations on the number of nested conditional preprocessor directives; reduce the depth of the conditional preprocessor directives nested in the program.
`Conflicting GRPDEFs: group name NAME.`	Link Error	Object module is probably corrupted.
`Conflicting argument declarations for function FUNCTION.`	Compile Error	Arguments of the named function prototype declaration do not match those for the old-style function definition of the same name; ensure that the function declaration matches that of the old-style function definition. A better course is to change the old-style function definition to a new-style definition that matches the function prototype declaration.
`Constant expression must be integer.`	Compile Error	Constant integer expression is expected in this context. Ensure the expression conforms to the semantics of a constant expression that computes an integer value.
`Conversion from TYPE to TYPE is compiler-dependent.`	Compile Warning	Avoid converting between a function pointer and other types of pointers, because you should not access functions as data, and you cannot execute data as functions.
`Could not allocate stack space. Try decreasing the Maximum stack size option in the Run Options dialog.`	Fatal Run-time Error	There is insufficient memory to allocate the Maximum Stack Size you have specified. LabWindows/CVI allocates the maximum size on the stack at the beginning of execution.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Could not find the DLL header file HEADER FILE.`	Glue Code Generation Error	LabWindows/CVI could not find the file that contains the prototypes for the functions in the DLL. When generating glue code, ensure that you specify the correct filename. When loading a DLL, ensure that a header file with the same base name as the DLL exists.
`d modifier not valid in Fmt/FmtOut/FmtFile.`	Non-Fatal Run-time Error	`d` modifier cannot be used in `Fmt`, `FmtOut`, or `FmtFile`.
`Declaration of NAME does not match previous declaration at POSITION.`	Compile Error	You declared a variable or function twice, and its type in the first declaration does not match its type in the second declaration.
`Declared parameter NAME is missing.`	Compile Error	Declaration for a parameter in an old style parameter list is missing, or the declaration does not match to any parameter name in the list. Ensure that the names in the old-style function definition have corresponding parameter declarations. A better course is to convert the old-style function definition to the new-style function definition requiring prototypes.
`"defined" expects an identifier argument.`	Compile Error	Preprocessor `defined()` operator requires a single identifier argument; ensure that you use an identifier and not an expression.
`Dereference of a NUMBER byte object where only NUMBER bytes exist.`	Fatal Run-time Error	Pointer expression you dereferenced points to an object that is smaller than the type of the dereference. For example, if an `int` pointer points to an object of type `char`, you cannot dereference the pointer because it points to only 1 byte, whereas an `int` requires 4 bytes.
`Dereference of data pointer used as a function.`	Fatal Run-time Error	You converted a data pointer to a function pointer and then dereferenced it. You can examine or modify data, but you cannot execute it as a function.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Dereference of function pointer used as data.`	Fatal Run-time Error	You converted a function pointer to a non-function pointer and then dereferenced it. You can only execute functions and access them as data.
`Dereference of invalid pointer expression.`	Fatal Run-time Error	Pointer expression you dereferenced is invalid. It is probably the result of a previous invalid pointer operation.
`Dereference of null pointer.`	Fatal Run-time Error	Pointer expression you dereferenced has the value `NULL` and cannot be dereferenced.
`Dereference of out-of-bounds pointer: NUMBER bytes (NUMBER elements) before start of array.`	Fatal Run-time Error	Pointer expression you dereferenced is invalid because it refers to a location before the start of an array. The error message shows the number of bytes and the number of array elements in the array. The expression is probably the result of previous illegal pointer arithmetic.
`Dereference of out-of-bounds pointer: NUMBER bytes (NUMBER elements) past end of array.`	Fatal Run-time Error	Pointer expression you dereferenced is invalid because it refers to a location past the end of an array. The error message shows the number of bytes and the number of array elements past the end of the array. The expression is probably the result of previous illegal pointer arithmetic.
`Dereference of pointer to freed memory.`	Fatal Run-time Error	Pointer expression you dereferenced is invalid because it refers to a location in dynamic memory that you deallocated with the `free` function. Once memory is free, all pointers into that block of memory become invalid.
`Dereference of unaligned pointer.`	Fatal Run-time Error [UNIX only]	Pointer expression you dereferenced is invalid because it points to an address that does not have the proper alignment for the type of the dereferenced object. SPARCstation architecture requires that 16-bit objects be halfword aligned, 32-bit objects be word aligned, and 64-bit objects be doubleword aligned.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Dereference of uninitialized pointer.`	Fatal Run-time Error	Pointer expression you dereferenced is invalid because you did not initialize it. It is probably an uninitialized local variable. You must initialize local variables before you use them.
`Duplicate case label NAME.`	Compile Error	Case label value appears more than once in the switch statement. Eliminate any duplicate case label values in the switch statement.
`Duplicate definition for NAME previously declared at POSITION.`	Compile Error	You redeclared a previously defined parameter name; eliminate one of the parameter declarations.
`Duplicate field name NAME in TYPE.`	Compile Error	You have already declared the member name of the `struct` or `union` type. Eliminate one of the member declarations from the struct or union type declaration.
`Dynamic memory is corrupt.`	Fatal Run-time Error	LabWindows/CVI encountered corrupt data while allocating or freeing dynamic memory.
`Empty declaration.`	Compile Error or Warning	You did not declare an object or type. It is an error if the empty declaration appears in the context of an old-style parameter declaration.
`Elf library is out of date.`	Object Load Error	LabWindows/CVI expects a more recent version of the shared library (`libelf.so`) that it uses to load ELF objects. As a result, LabWindows/CVI is unable to read or write object and library files.
`'enum NAME' declared inside parameter list has scope only for this declaration.`	Compile Warning	Enumeration you declared in the parameter list has scope only within the parameter list. As a result, its type is incompatible with all other types. You must declare the enumeration type before you declare function types that use it.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Error at or near character NUMBER in the format string: STRING.`	Non-Fatal Run-time Error	Error exists in the format string at index *NUMBER*. *NUMBER* is 1-based.
`Error in Elf Library encountered while reading external module: NAME.`	Object Load Error	Object module is corrupted or is of a type that LabWindows/CVI cannot load.
`Error: compiling FILE for DLL exports.`	DLL Import Library Creation Error	When creating a DLL using the Include File method for specifying exported symbols, an error occurred while compiling the include file.
`Error: Incompatible type for function or variable NAME in header FILE used to specify exports.`	DLL Link Error	When creating a DLL using the Include File method for specifying exported symbols, the type of the symbol in the include file did not match the type in the source file.
`Expecting an enumerator identifier.`	Compile Error	Compiler expects an enumeration constant identifier after the opening `{` in an `enum` type declaration.
`Expecting an identifier.`	Compile Error	Compiler expects an identifier in the current syntactic context. Check the syntax of the declaration, statement, or preprocessor directive.
`Expecting integer constant, push, or pop.`	Compile Error	`pack` pragma requires at least one parameter.
`Extra default label.`	Compile Error	`default` label has already appeared for this switch statement. Eliminate the extraneous `default` label.
`Extraneous 0-width bit field TYPE NAME ignored.`	Compile Warning	Named bit field has no width and therefore has no storage allocated to it.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Extraneous formal parameter specification.`	Compile Error	This error occurs when the compiler is processing what it assumes to be an old-style function declaration and encounters what it assumes to be the function's parameter names. If this is an old-style function declaration, make sure that the parameter names appear only in the function definition and not in any declaration of that function. If this is a new-style function declaration (prototype), then probably the identifier that the compiler assumes to be a parameter name is really a typedef name. Make sure that you previously declared the identifier as a typedef.
`Extraneous identifier NAME.`	Compile Error	Identifier appears in a context where the compiler expects a type name, such as in a cast operation or as the operand of `sizeof()`. Syntactically, a type name is a declaration of a function or an object of that type that omits the identifier.
`Extraneous return value.`	Compile Error	Return statement appears in a void function and therefore no return value is necessary; eliminate the expression from the return statement.
`Failed to load DLL FILE.`	Link Error	LabWindows/CVI could not find the DLL. Ensure that it is in one of the default directories searched by Microsoft Windows, or that it includes a complete path name.
`Failed to open external module.`	Object Load Error	LabWindows/CVI could not open the external module for loading. Ensure that the external module has read access and that you did not inadvertently rename or delete it.
`Field name expected.`	Compile Error	The compiler expects an identifier to follow a `.` or –>.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Field name missing.`	Compile Error	Identifier is missing from a member (field) declaration in a `struct` or `union` type declaration. Make sure that an identifier follows the member type specifier.
`Format string integer is too big.`	Non-Fatal Run-time Error	Integer used in the format string is too large.
`Found TYPE expected a function.`	Compile Error or Warning	In an expression, the compiler expects the name of a function or pointer to function to precede a `(`. In a `#pragma` line, the compiler expects the name of a function after the `pragma` type.
`Function definitions are not allowed in the interactive window.`	Compile Error	Function definitions cannot appear in the Interactive Execution window.
`Function FUNCTION: (STRING == NUMBER).`	Non-Fatal Run-time Error	Library function could not perform its task. The integer *`NUMBER`* is either the function return value or the value of a global variable that explains why the function failed. Refer to the library function reference material for more information about the error.
`Function FUNCTION has an unsupported return type size.`	Glue Code Generation Error	Glue code generation or the DLL loading facilities do not support the return type of the function.
`Function requires extra code to handle Callbacks: FUNCTION.`	Glue Code Generation Error	Automatic glue code generation facility cannot generate complete code for this function because one of its parameters is a function pointer or it returns a function pointer. You must generate and modify the glue source code.
`h modifier is only valid with d, i, n, o, u, and x specifiers.`	Non-Fatal Run-time Error	You can only use the `h` modifier with integer format specifiers.
`Header name literal too long.`	Compile Error	Header name length exceeds implementation limitations. Ensure that the header name is properly terminated with a `>` or a `"`, or shorten the string literal.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Ill-formed constant integer expression.`	Compile Error	Constant integer expression that appears in a preprocessor directive is syntactically invalid. Check the expression for trailing tokens.
`Ill-formed hexadecimal escape sequence \xCHAR.`	Compile Error	Ensure that a hexadecimal character ([0–9, a–f, or A–F]) follows the `\x` escape sequence introduction.
`Ill-formed hexadecimal escape sequence.`	Compile Error	Ensure that a hexadecimal character (0–9, a–f, or A–F]) follows the `\x` escape sequence introduction.
`Illegal argument(s) to library function.`	Fatal Run-time Error	One or more of the arguments to the library function are invalid. Refer to the library documentation for the function.
`Illegal case label.`	Compile Error	Case label appears outside the context of a switch statement. Remove the case label.
`Illegal character CHAR.`	Compile Error	Character or character escape sequence outside the legal character set for an ANSI C source file appears in a context other than a character string or character literal.
`Illegal continue statement.`	Compile Error	`continue` statement appears outside a loop statement. Remove the `continue` statement.
`Illegal default label.`	Compile Error	`default` label appears outside the context of a switch statement. Remove the `default` label.
`Illegal expression.`	Compile Error	Compiler encountered the wrong type of token while parsing an expression where it expected an identifier, string literal, integer constant, floating constant, or `(`.
`Illegal extern definition of NAME; all interactive window variable definitions must be static.`	Compile Error	No Interactive Execution window definitions are visible outside the scope of the Interactive Execution window. You cannot initialize external symbols in the Interactive Execution window.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Illegal formal parameter types.`	Compile Error	Parameter type of void appears in a function prototype declaration that has more than one argument. Remove the void parameter type or change the function prototype so that it contains only the single void parameter type.
`Illegal header name; #include expects "FILE" or <FILE>.`	Compile Error	Unexpected character follows an `#include` where a header filename of the form `"FILE"` or `<FILE>` is expected. It is also possible that the header filename beginning quote character is different than the expected closing quote character, such as `<FILE"`.
`Illegal initialization for NAME.`	Compile Error	Ensure that the initialization is not for a function declaration rather than a pointer to a function.
`Illegal initialization for parameter.`	Compile Error	Parameter declarations cannot have default value initializations in ANSI C. Eliminate the initialization.
`Illegal initialization for parameter NAME.`	Compile Error	Parameter declarations cannot have default value initializations in ANSI C. Eliminate the initialization.
`Illegal initialization of extern NAME.`	Compile Error	You attempted to initialize an extern declaration that appears in a local scope. Eliminate the initialization.
`Illegal return type TYPE.`	Compile Error	Function is declared with an illegal return type, or a return statement expression type is not the same as the return type of the function in which it appears. If the diagnostic is for a function declaration, ensure that the return type is not an array type or a function type. If the diagnostic is for a return statement, the containing function is probably declared void and can contain no expression in its return statement.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Illegal return type; found TYPE expected TYPE.`	Compile Error	Return statement expression type is not the same as the return type of the function in which it appears. Ensure that the type of the return expression is consistent (assignment compatible) with the function return type.
`Illegal separator character or illegal position of separator character.`	Non-Fatal Run-time Error	Either the separation characters < and > were not present in the format string, or they were in the wrong place.
`Illegal source filename specified for #line; s-char-sequence expected.`	Compile Error	Only token that can follow the line number specification in a `#line` preprocessor directive is an optional string literal specifying a source filename. A sequence of tokens also can follow the `#line` token if, after the compiler performs macro expansion on the source line, the source line conforms to one of the two allowable forms of `#line` preprocessor directives: `#line` *`line-number-digit-sequence`* `#line` *`line-number-digit-sequence "filename"`*
`Illegal statement termination.`	Compile Error	During compilation of a sequence of statements, the compiler encountered a token that it expected either to begin a new statement, begin an `else` clause of an `if` statement, be a statement label, be a case label, or terminate a compound statement, such as `}`. Depending on the context of the location of where the compiler issued the diagnostic, ensure that the statement syntax is correct for the cases listed above.
`Illegal type array of TYPE.`	Compile Error	You attempted to declare an array of functions. You probably intended to declare an array of function pointers instead.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Illegal type const TYPE.`	Compile Error	You used more than one qualifier, such as `const` or `volatile`, in a type specification; for example, `const const int`. Do not use the `const` and `volatile` qualifiers more than once each in the same type.
`Illegal type for symbol 'DllMain': TYPE.`	Compile Error	`DllMain` does not conform to the accepted prototype. `int__stdcall DllMain (HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved);`
`Illegal type for symbol 'WinMain': TYPE.`	Compile Error	`WinMain` does not conform to the accepted prototype. `int__stdcall WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);`
`Illegal type volatile TYPE.`	Compile Error	You used more than one qualifier, such as `const` or `volatile`, in a type specification; for example, `const const int`. Do not use the `const` and `volatile` qualifiers more than once each in the same type.
`Illegal use of type name NAME.`	Compile Error	You used a typedef name in the context of a primary expression. If you intended to use a type cast, parenthesize the typedef name. Otherwise you must use a macro name, enumeration constant, variable name, or function name in this context.
`Illegal value matched to asterisk.`	Non-Fatal Run-time Error	Integer argument that matches to an asterisk (*) in the format string has an invalid value given the context in which it appears.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Illegal variable declaration; only static and extern variable classes are valid in the interactive window.`	Compile Error	Change the variable declaration to be either `static` or `extern`.
`Import Variables cannot be used in global variable initialization.`	Compile Error	You used a global variable marked as `__import` or `declspec(dllimport)` in an initializer of another variable.
`Include files nested too deeply.`	Compile Error	Number of nested `#include` files exceeds compiler limits. Reduce the number of nested `#include` preprocessor directives.
`Inconsistent linkage for NAME previously declared at POSITION.`	Compile Error	Current declaration of the identifier is inconsistent with a previous declaration of the same identifier with regard to linkage. Ensure that all declarations of the identifier that you intend to be `static` do not conflict with declarations without the `static` keyword in the same scope.
`Inconsistent type declarations for external symbol NAME in modules FILE1 and FILE2.`	Link Error	You declared two or more external symbols with the same name but not the same type. Check each program file that contains an external declaration of the symbol for type consistency.
`Initializer exceeds bit-field width.`	Compile Warning	Number of bits necessary to represent the initialization value of a bit field exceeds its declared width. The compiler truncates the initialization value to fit the bit field. The initialization value must be smaller or the bit field declaration must be wider.
`Initializer must be constant.`	Compile Error	Initializer must be an expression that conforms to the semantics for a constant expression.
`Insufficient number of arguments to FUNCTION.`	Compile Error	Function expects more arguments than you passed to it. Check the function declaration for the number of parameters to the function.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Insufficient system memory for Interactive Window`	Link Error	There is not enough memory to run the code in the Interactive Execution window.
`Insufficient system memory for project.`	Link Error	There is not enough memory to link the project.
`Insufficient user data memory for project.`	Link Error	There is not enough memory to link the project.
`Invalid hexadecimal constant.`	Compile Error	A token the compiler assumes to be a hexadecimal constant is badly formed. Ensure that token conforms to the syntax for hexadecimal constants, especially that a valid hexadecimal digit follows the `0x` or `0X` prefix.
`Invalid initialization type; found TYPE expected TYPE.`	Compile Error	Expression that initializes the object declaration is type incompatible with the object. Ensure that the initialization expression is assignment compatible with the object type. Ensure that all constituent values of an aggregate expression match the corresponding positional types of the aggregate type, such as member types of a `struct` or `union` type.
`Invalid octal constant.`	Compile Error	A token the compiler assumes to be an octal constant is badly formed. Ensure that the token conforms to syntax for octal constants, especially that a valid octal digit follows the leading `0` prefix.
`Invalid operand of unary &; NAME is declared register.`	Compile Error	It is illegal to take the address (`&` prefix operator) of an object you declare to be of `register` class. Remove the `register` keyword from the object declaration if you want to apply the address operator to it.
`Invalid pointer argument to library function.`	Fatal Run-time Error	Pointer expression you passed to the library function is invalid. It is probably the result of a previous invalid pointer operation.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Invalid size for a real.`	Non-Fatal Run-time Error	4 and 8 are the only valid sizes that you can specify with the `b` modifier for the `f` (real) specifier.
`Invalid size for an integer.`	Non-Fatal Run-time Error	1, 2, and 4 are the only valid sizes that you can specify with the `b` modifier for the `i`, `d`, `x`, `o`, and `c` modifiers.
`Invalid storage class.`	Compile Error	`extern` is the only allowable explicit storage class specifier for a function declaration that has block scope.
`Invalid struct field declarations.`	Compile Error	Compiler encountered an invalid token while processing a struct or union type declaration. The compiler expected a token that begins a member type specifier where the type specifier is one of `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`, <*`struct-or-union-specifier`*>, <*`enum-specifier`*>, or <*`typedef-name`*>.
`Invalid type argument TYPE to sizeof.`	Compile Error	You applied `sizeof` operator to a function type or incomplete struct or union type. A function type has no size, and the size of an incomplete struct or union type is unknown before its full declaration.
`Invalid type specification.`	Compile Error	Combination of type specifiers is incompatible. You can use the type specifier `short` only in combination with `int`. You can use the type specifier `long` only in combination with `int` and `double`. You can use the type specifiers `signed` and `unsigned` only in combination with one of the basic integer types (`char`, `short`, `int`, `long`).

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Invalid union field declarations.`	Compile Error	Compiler encountered an invalid token while processing a struct or union type declaration. The compiler expected a token that begins a member type specifier where the type specifier is one of `void`, `char`, `short`, `int`, `long`, `float`, `double`, `signed`, `unsigned`, <*`struct-or-union-specifier`*>, <*`enum-specifier`*>, or <*`typedef-name`*>.
`Invalid use of TOKEN.`	Compile Error	This error occurs during compilation of a type specification. The specified *`TOKEN`* is not valid in the context of the type specifier. Two common errors are use of a storage class other than `register` for a parameter declaration and using the storage class `register` for a global object declaration.
`l format specifier not valid in Fmt/FmtOut/FmtFile.`	Non-Fatal Run-time Error	You can use the `l` format specifier only in `Scan`, `ScanOut`, and `ScanFile`.
`l modifier is only valid with d, i, n, o, u, and x specifiers.`	Non-Fatal Run-time Error	`l` format specifier is valid only for integer format specifiers.
`L modifier is only valid with e, f, and g specifiers.`	Non-Fatal Run-time Error	`L` modifier, which specifies that the argument is a long double, can be used only in the floating point formats.
`l modifier is only valid with e, f, g, d, i, n, o, u, and x specifiers.`	Non-Fatal Run-time Error	`l` format specifier is valid only for integer and real format specifiers.
`Left operand of –> has incompatible type TYPE.`	Compile Error	Left operand of the –> dereference operation is either not a pointer to `struct` or `union` type, or it is not a pointer type at all.
`Left operand of . has incompatible type TYPE.`	Compile Error	Left operand of the `.` member selection operation must be a `struct` or `union` type.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Library function error (STRING == NUMBER).`	Non-Fatal Run-time Error	Library function could not perform its task. The integer *NUMBER* is either the function return value or the value of a global variable that explains why the function failed. Refer to the library function reference material for more information about the error.
`Lvalue required.`	Compile Error	An lvalue is required in this context. Ensure that the expression conforms to the semantics of an lvalue.
`Macro expansion too large.`	Compile Error	Macro expansion has exceeded the compiler implementation size limitation.
`Macro parameter must follow # operator.`	Compile Error	# operator requires that a macro parameter immediately follow it in a macro replacement list.
`Matching push not encountered or already popped.`	Compile Error	`pack` pragma used a named pop that does not balance with the push of the same name.
`Missing { in initialization of TYPE.`	Compile Error	Initialization of a `struct`, `union`, or `array` type, is missing a starting `{` for an aggregate initialization value.
`Missing #endif`	Compile Error	`#if`, `#ifdef` preprocessor directive must have a corresponding `#endif` in the same source file.
`Missing #include file name; #include expects "FILE" or <FILE>.`	Compile Error	No include filename follows the `#include` preprocessor directive. Ensure that a filename of the correct form follows `#include` or that any macro that follows `#include` expands into the correct form for an include filename.
`Missing '.`	Compile Error	Termination single quote character `'` is missing from a character or wide character literal.
`Missing CHAR.`	Compile Error	Check for unterminated string or character literal.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Missing argument to variable argument function.`	Fatal Run-time Error	Variable argument function requires at least one argument beyond the last formal parameter.
`Missing array size.`	Compile Error	You attempted to define a block scope object or type that is an array which has an element type that is an incomplete array type, such as an array with unspecified size. The array element type must be a complete array type, such as an array type with a known size.
`Missing format string integer.`	Non-Fatal Run-time Error	Integer that corresponds to an asterisk in a format string is missing. Incorrect ordering of the arguments can cause this. This integer must precede the actual argument.
`Missing identifier.`	Compile Error	Identifier that specifies the object name is missing from the object declaration. Ensure that an identifier follows the object type specifier.
`Missing label in goto.`	Compile Error	`goto` statement is missing an identifier label.
`Missing parameter name to function FUNCTION.`	Compile Error	Parameter list of the function definition is missing an identifier for one of its parameter declarations. All parameter declarations for a function definition must include an identifier except for the special case of a parameter list consisting of a single parameter of type `void`, in which there must not be an identifier.
`Missing parameter type.`	Compile Error	Type specifier is missing from a parameter declaration in a new-style (prototype) function declaration. Ensure that the function declaration is not mixing old-style parameter declarations with new-style (prototype) declarations.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Missing prototype.`	Compile Error	Function declaration or call is for a function without prototype declaration information. The compiler issues the diagnostic if the Require Function Prototypes compiler option is enabled.
`Missing return value.`	Fatal Run-time Error	Function does not return a value, although you declared it with a return type. If you did not intend for the function to return a value, you must declare it as a `void` function. Otherwise, you must use a `return` statement to return a value.
`Missing return value.`	Compile Warning	Non-void function does not return a value. Add a return statement with an expression of the function return type. The compiler issues the diagnostic if the Require Return Value for Non-void Functions compiler option is enabled.
`Missing right bracket (]).`	Fatal Run-time Error	Format string has mismatched brackets.
`Missing struct tag.`	Compile Error	Tag name is missing from an incomplete struct or union declaration.
`Missing terminating null in string argument.`	Fatal Run-time Error	Library function expects a string argument, but the argument you passed points to an array of characters that is not null-terminated.
`Missing union tag.`	Compile Error	Tag name is missing from an incomplete struct or union declaration.
`Multiply defined symbol NAME in modules FILE1 and FILE2.`	Link Error	The files being linked contain more than one definition for *NAME*.
`Naked functions are not supported.`	Compile Error	LabWindows/CVI does not work with the `naked` keyword.
`NAME is a predefined macro and cannot be the subject of an #undef.`	Compile Error	Make sure that the name you specify for the `#undef` preprocessor directive is not that of a predefined macro.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`No data relocation section found for external module: FILE.`	Link Error	External object module does not contain the relocation information necessary to link it in with the rest of the project. You cannot load an executable as an object module.
`No data section found for external module: FILE.`	Link Error	External object module does not contain the initialized data necessary to link it in with the rest of the project. Ensure that you built the external object file correctly.
`No pack settings currently pushed.`	Compile Error	`pack` pragma used a pop when there were no pushes.
`No symbol table found for external module: FILE.`	Link Error	External object module does not contain the symbol table information necessary to link it in with the rest of the project. Ensure that you built the external object file correctly.
`No text relocation section found for external module: FILE.`	Link Error	External object module does not contain the relocation information necessary to link it in with the rest of the project. You cannot load a linked executable as an object module.
`No text section found for external module: FILE.`	Link Error	External object module does not contain the initialized instruction data necessary to link it in with the rest of the project. Ensure that you built the external object file correctly.
`Non-terminated address list.`	Fatal Run-time Error	You attempted to pass an address list array that you did not terminate with `-1` to a GPIB-488.2 function that expects the array to terminate with `-1`.
`Not enough parameters.`	Non-Fatal Run-time Error	Number of arguments the format string expects is more than the number of arguments you passed in.
`Not enough space for casting expression to TYPE.`	Non-Fatal Run-time Error	Block of memory you obtained from `malloc` or `calloc` is not large enough for a single object of type *`TYPE`* and cannot be cast to that type.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Null Pointer.`	Fatal Run-time Error	Pointer expression you passed to the library function has the value `NULL`, which is not a valid value for the function.
`Null pointer argument to library function.`	Fatal Run-time Error	Pointer expression you passed to the library function has the value `NULL`, which is not a valid value for the function.
`NUMBER is an illegal array size.`	Compile Error	Make sure that the size of the array declaration is > 0.
`NUMBER is an illegal bit field size.`	Compile Error	Make sure that the size you specified for the bit field is ≥ 0 and ≤ 32.
`NUMBER line(s) truncated. File set to read-only.`	Compile Error	Occurs when reading in source or include file. Lines are limited to 254 characters, where tabs count as 1. Use the editor in which you created the file to split the line.
`Number of arguments exceed the maximum supported.`	Non-Fatal Run-time Error	Number of arguments exceeds the maximum that the formatting functions support.
`Number of points is too large for current waveform buffer.`	Fatal Run-time Error	Message appears when the `numberofPoints` parameter of a data acquisition waveform generation function is larger than the `numberofPoints` parameter to the function which set up the waveform buffer.
`Object module contains unsupported FAR pointers.`	Object Load Error	External object module contains `FAR` pointers, which you cannot implement in LabWindows/CVI.
`One of the arguments to FUNCTION has an unsupported size.`	Glue Code Generation Error	One of the function arguments has a type that the LabWindows/CVI glue code generation and DLL loader do not support.
`Only object modules produced by WATCOM C 386 fully supported.`	Link Error	External object module contains OMF records that LabWindows/CVI does not recognize or support. Ensure that the object file was compiled with a Watcom C 386 compiler with the recommended options.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Operands of '=' have incompatible calling conventions.`	Compile Error	Function pointer is assigned an expression that does not match its calling convention.
`Operands of [one from set of binary operators] have illegal types TYPE and TYPE.`	Compile Error	Types of the two operands to the binary operator are illegal according to the ANSI C standard.
`Operands of [one from set of binary operators] have incompatible types.`	Compile Error	Types of the two operands to the binary operator are not compatible according to the ANSI C standard.
`Operand of unary OPERATOR has illegal type TYPE.`	Compile Error	Type of the operand to the unary operator is not valid.
`Out-of-bounds pointer argument (before start of array).`	Fatal Run-time Error	Pointer expression you passed to the library function is invalid because it refers to a location that is before the start of an array. The expression is probably the result of previous illegal pointer arithmetic.
`Out-of-bounds pointer argument (past end of array).`	Fatal Run-time Error	Pointer expression you passed to the library function is invalid because it refers to a location past the end of an array. The expression is probably the result of previous illegal pointer arithmetic.
`Out-of-bounds pointer arithmetic: NUMBER bytes (NUMBER elements) before start of array.`	Non-Fatal Run-time Error	Pointer arithmetic expression is invalid because the resulting value refers to a location before the start of an array. The error message shows the number of bytes and number of array elements before the beginning of the array.
`Out-of-bounds pointer arithmetic: NUMBER bytes (NUMBER elements) past end of array.`	Non-Fatal Run-time Error	Pointer arithmetic expression is invalid because the resulting value refers to a location past the end of an array. The error message shows the number of bytes and number of array elements past the end.
`Out of memory for user protection information.`	Fatal Run-time Error	Could not allocate memory required to store user protection information.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Overflow in constant CONSTANT.`	Compile Warning	Value of a constant or constant expression exceeds the limits of the type. Ensure that the value does not exceed the maximum value for the expression type.
`Overflow in constant expression.`	Compile Warning	Value of a constant or constant expression exceeds the limits of the type. Ensure that the value does not exceed the maximum value for the expression type.
`Overflow in floating constant CONSTANT.`	Compile Warning	Value of a constant or constant expression exceeds the limits of the type. Ensure that the value does not exceed the maximum value for the expression type.
`Overflow in hexadecimal escape sequence.`	Compile Warning	Value of a constant or constant expression exceeds the limits of the type. Ensure that the value does not exceed the maximum value for the expression type.
`Overflow in octal escape sequence.`	Compile Warning	Value of a constant or constant expression exceeds the limits of the type. Ensure that the value does not exceed the maximum value for the expression type.
`Overflow in value for enumeration constant CONSTANT.`	Compile Error	Value of a constant or constant expression exceeds the limits of the type. Ensure that the value does not exceed the maximum value for the expression type.
`Overflow occurred during the conversion of the int. The absolute value is too big for the size.`	Non-Fatal Run-time Error	Number was too large to store in the integer of the specified size.
`Overflow occurred during the conversion of the float. The number is too big for type float.`	Non-Fatal Run-time Error	Number was too large to store in a 4-byte real.
`Overflow occurred during the conversion of the int. The signed value is too big for the size.`	Non-Fatal Run-time Error	Number was too large to store in the integer of the specified size.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Pack pragma valid values are 1, 2, 4, 8, and 16.`	Compile Error	`pack` pragma alignment value parameter must be `1`, `2`, `4`, `8`, or `16`.
`Parameter type incompatible with format specifier.`	Non-Fatal Run-time Error	Parameter type is not compatible with the type that the format string expects. An argument is either missing or of the wrong type.
`Parameter type mismatch: expecting TYPE but found TYPE.`	Non-Fatal Run-time Error	Parameter type does not match the type that the format string expects. The arguments might not be in the right order, or an argument might be missing.
`Pointer arithmetic involving invalid pointer.`	Non-Fatal Run-time Error	Pointer arithmetic expression is invalid because one of the subexpressions contains an invalid pointer.
`Pointer arithmetic involving null pointer.`	Non-Fatal Run-time Error	Pointer arithmetic expression is invalid because one of the subexpressions contains the value `NULL`.
`Pointer arithmetic involving pointer to freed memory.`	Non-Fatal Run-time Error	Pointer arithmetic expression is invalid because one of the subexpressions contains a pointer to dynamic memory that you deallocated with the `free` function. Once memory is free, all pointers into that block of memory are invalid.
`Pointer arithmetic involving pointer to function.`	Non-Fatal Run-time Error	Pointer arithmetic expression is invalid because one of the subexpressions is a function pointer.
`Pointer arithmetic involving uninitialized pointer.`	Non-Fatal Run-time Error	Pointer arithmetic expression is invalid because one of the subexpressions contains an uninitialized pointer. It is probably an uninitialized local variable.
`Pointer comparison involving address of nonarray object.`	Non-Fatal Run-time Error	One of the pointer expressions in the comparison is invalid because it does not point into an array. Both expressions in pointer comparisons must point into the same object.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Pointer is invalid.`	Non-Fatal Run-time Error	Pointer argument to the function contains an invalid address.
`Pointer points to freed memory.`	Non-Fatal Run-time Error	Pointer argument to the function points to memory that you already freed.
`Pointer subtraction involving address of nonarray object.`	Non-Fatal Run-time Error	One of the pointer expressions in the subtraction is invalid because it does not point into an array. Both expressions in pointer subtractions must point into the same object.
`Pointer to a local is an illegal return value.`	Compile Error	Value returned from the function is a pointer to a parameter or local variable. Because the lifetime of a parameter or local variable ends when you return from the function, any pointer to such an object is invalid.
`Pointer to a parameter is an illegal return value.`	Compile Error	Value returned from the function is a pointer to a parameter or local variable. Because the lifetime of a parameter or local variable ends when you return from the function, any pointer to such an object is invalid.
`Pointer to free memory passed to library function.`	Fatal Run-time Error	Pointer expression you passed to the library function is invalid because it refers to a location in dynamic memory that you deallocated with the function `free`. Once memory is free, all pointers into that block of memory are invalid.
`pragma pack(pop...) does not set alignment. Use separate pack pragma.`	Compile Warning	You used a pragma pop with an alignment value. Use separate `pack` pragmas for popping and setting the alignment value.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Project not linked.`	Link Warning	This error occurs when the compiler reports one or more link errors in the Interactive Execution window and the project is not in a linked state. This warning provides a possible explanation for the link errors. The Interactive Execution window does not link to the project unless the project is in a linked state. If you are referencing project symbols from the Interactive Execution window, use the **Build Project** command from the **Build** menu to compile and link the project first.
`Qualified function type ignored.`	Compile Warning	Any qualification of a function declaration is extraneous but harmless.
`Read error.`	Link Error	Error has occurred while attempting to read a file. Ensure that the file has access permission and that it is in the correct format.
`Redeclaration of '%s' with different calling convention, previously declared at %w.`	Compile Error	Function has been redeclared with a different calling convention.
`Redeclaration of macro parameter NAME.`	Compile Error	Parameter name has already been used once by the macro. Choose another parameter name.
`Redeclaration of NAME.`	Compile Error	Declared name conflicts with a previous declaration in the same scope and name space. You have already used the name in this scope. Choose another name for this declaration.
`Redeclaration of NAME previously declared at POSITION.`	Compile Error	Declared name conflicts with a previous declaration in the same scope and name space. You have already used the name in this scope. Choose another name for this declaration.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Redefinition of label NAME previously defined at POSITION.`	Compile Error	You already used the statement label in this function. A statement label must be unique within the function in which you use it.
`Redefinition of macro NAME.`	Compile Error	Macro has already been defined with a replacement list different from the current definition. The same macro definition for a name may appear in the same file more than once as long both definitions agree in name and number of parameters and their replacement lists are identical.
`Redefinition of NAME previously defined at POSITION.`	Compile Error	You have already defined the object or function in the current scope. Eliminate one of the two definitions.
`Reference parameter expected.`	Non-Fatal Run-time Error	Function expected a pointer but you passed it a scalar.
`Register declaration ignored for TYPE NAME.`	Compile Warning	`register` storage class conflicts with the semantics of the type declared for the object. If you declared the object to be of an array, `struct`, or `union` type, or you qualified it as `volatile`, remove the `register` keyword from the declaration.
`Register declaration ignored for TYPE parameter.`	Compile Warning	`register` storage class conflicts with the semantics of the type you declared for the parameter prototype. If you declared the object to be of `struct` or `union` type, or you qualified it as `volatile`, remove the `register` keyword from the prototype parameter declaration.
`Repeat value not valid with s/l format specifiers.`	Non-Fatal Run-time Error	You cannot use a repeat value with the `s` and `l` format specifiers.
`Result of unsigned comparison is constant.`	Compile Warning	Result of <*UNSIGNED INTEGER EXPRESSION*> ≥ 0 always evaluates to 1.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Segment must be of class CODE, DATA, BSS, or STACK: segment name NAME.`	Load Error	External object module contains an unknown segment class. Object modules must not contain any specially-named segments.
`Segment must be USE32: segment name NAME.`	Link Error	External object module you loaded contains unsupported 16-bit segments. LabWindows/CVI supports only 32-bit object modules. Ensure that the external object module was compiled with a 32-bit compiler.
`'signed' type mismatch between TYPE and TYPE.`	Compile Warning	This warning is issued when the signs of the lvalue and rvalue expressions in a pointer assignment operation do not agree. Both lvalue and rvalue are pointers to integer types but they point to integer types of differing signs, which might cause problems if you later dereference the lvalue. This diagnostic is issued if you select the Enable Signed/Unsigned Pointer Mismatch Warning compiler option.
`Simple/Array conflict with format specifier.`	Non-Fatal Run-time Error	Array you passed to the function matches to a format specifier for a scalar, or a scalar you passed to the function matches to a format specifier for an array.
`Size of array of TYPE exceeds SIZE bytes.`	Compile Error	Size of the array or struct/union type exceeds the compiler limitation of `INT_MAX` bytes.
`Size of TYPE exceeds SIZE bytes.`	Compile Error	Size of the array or struct/union type exceeds the compiler limitation of `INT_MAX` bytes.
`sizeof applied to a bit field.`	Compile Error	Do not use the `sizeof()` operation on a bit-field.
`Specified width is too small to read the number.`	Non-Fatal Run-time Error	Width you specified for a format specifier was not large enough to contain a complete number. Example: you specify a width of `2` for a float, and the number is `-.02` ; the negative sign and decimal point do not constitute a valid number.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Stack Overflow.`	Fatal Run-time Error	Program exceeds the stack limit. Change the size of the stack in the Run Options dialog box, if you think that the code is executing correctly. Otherwise, ensure that the program does not contain any infinite recursion.
`'struct NAME' declared inside parameter list has scope only for this declaration.`	Compile Warning	Structure declared in the parameter list has scope only within the parameter list. As a result, its type is incompatible with all other types. You must declare the structure type before you declare function types that use it.
`Structures containing unspecified size array fields must contain other fields.`	Compile Error	Structures that contain arrays with unspecified size must contain at least one other non-zero size member. LabWindows/CVI supports these types of structures as an extension to the ANSI C standard.
`Subtraction involving invalid pointer.`	Non-Fatal Run-time Error	One of the pointer expressions in the subtraction is invalid. It is probably the result of a previous invalid pointer operation.
`Subtraction involving null pointer.`	Non-Fatal Run-time Error	One of the pointer expressions in the subtraction has the value `NULL`. Both expressions in pointer subtractions must point into the same array object.
`Subtraction involving uninitialized pointer.`	Non-Fatal Run-time Error	One of the pointer expressions in the subtraction is invalid because you did not initialize it.
`Subtraction of pointers to different objects.`	Non-Fatal Run-time Error	Pointer expressions in the subtraction point to two distinct objects. Both expressions in pointer subtractions must point into the same array object.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Subtraction of pointers to freed memory.`	Non-Fatal Run-time Error	One of the pointer expressions in the subtraction is invalid because it refers to a location in dynamic memory that you deallocated with the function `free`. Once memory is free, all pointers into that block of memory are invalid.
`Switch statement with no cases.`	Compile Warning	Switch statement contains no `case` or `default` label.
`Symbol NAME defined in modules FILE and FILE. In Borland mode, multiple modules must not contain uninitialized definitions of the same global variable. Borland creates a separate variable for each definition. LabWindows/CVI and other linkers resolve all definitions to the same variable. If you want separate variables, use different names or the "static" keyword. If you want one variable, change all definitions except one to "extern" declarations.`	Link Error	In Borland mode, multiple modules must not contain uninitialized definitions of the same global variable. Borland creates a separate variable for each definition. LabWindows/CVI and other linkers resolve all definitions to the same variable. If you want separate variables, use different names or the “static” keyword. If you want one variable, change all definitions except one to “extern” declarations.
`Symbol NAME exported from header FILE not found in DLL.`	DLL Link Error or Import Library Creation Error.	When you used the Include File method for specifying the symbols to export from a DLL, one of the symbols you declared in the include file was not in the DLL project. Or, when you created import libraries from an include file and a DLL, one of the symbols you declared in the include file was not in the DLL.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Syntax error; found TOKEN1 expecting TOKEN2.`	Compile Error	Syntax error occurred because the compiler found `TOKEN1` instead of `TOKEN2`.
`The __cdecl calling convention is not supported with functions returning floats, doubles, or structures in WATCOM Compatibility Mode.`	Compile Error	Function with an explicit `__cdecl` qualifier returns a `double`, `float` or structure, and your current compatible compiler is Watcom. Either remove the qualifier or change the function.
`The callback function, NAME, differs only by a leading underscore from another function or variable. Change one of the names for proper linking.`	Non-Fatal Run-time Error	When trying to match a callback name you specified in a `.uir` file to the callback function, the compiler found two symbols that are the same except for a leading underscore. Resolve this ambiguity by changing one of the names.
`Thread data is not supported.`	Compile Error	You cannot implement thread-local storage in LabWindows/CVI.
`Too many arguments to FUNCTION.`	Compile Error	Declaration for function *`FUNCTION`* contains fewer parameters than the number of arguments you passed in this function call.
`Too many arguments to variable argument function.`	Non-Fatal Run-time Error	You passed more arguments to the variable argument function than it expected. The extra arguments do not affect the function call in any way.
`Too many function parameters.`	Compile Error	Number of parameter declarations exceeds compiler limitations. Declare the function with fewer parameters.
`Too many initializers.`	Compile Error	Size of the initializer exceeds the size of the object. Ensure that the initializer matches the number/size of the object type.
`Too many macro parameters.`	Compile Error	Number of parameter declarations exceeds compiler limitations. Declare the macro with fewer parameters.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Too many parameters.`	Non-Fatal Run-time Error	Number of parameters you passed to a function exceed the number of parameters the format string expects.
`Type error in argument %d to %s, calling convention mismatch.`	Compile Error	Function or function pointer you passed to a function does not have the correct calling convention.
`Type error in argument NUMBER to NAME; TYPE is illegal.`	Compile Error	Argument you passed is an illegal array type or an incomplete type of which the size is unknown. Ensure that the argument is of a complete type.
`Type error in argument NUMBER to NAME; found TYPE expected TYPE.`	Compile Error	You passed an argument that is not type compatible with the prototype declaration for the parameter in that position. Ensure that the actual argument is type compatible with the parameter declaration.
`Type error: pointer expected.`	Compile Error	Expression you dereferenced with the '`*`', '`->`' or '`[]`' operator does not have pointer type.
`TYPE is an illegal bit field type.`	Compile Error	Only `int` and `unsigned` types are valid for bit field declarations; ensure that you use one of these types.
`TYPE used as an lvalue.`	Compile Warning	Type that cannot be modified is used as the target of an assignment. This was probably caused by an lvalue that is a dereference of an object declared as (`void *`).
`Unclosed comment.`	Compile Error	Comment is missing the closing `*/` delimiter.
`Undeclared identifier NAME.`	Compile Error	You did not previously declare *NAME*. You must declare all names before use. Ensure that you did not conditionally exclude *NAME* from compilation.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Undefined label NAME.`	Compile Error	You used the label *NAME* as the target of a `goto` statement in the function but it never appears as a statement label. Ensure the label appears in the same function as the `goto` statements of which it is a target. Non-local `goto` statements are illegal.
`Undefined size for TYPE NAME.`	Compile Error	You have defined an object with an incomplete type. Because the size of an incomplete type is unknown, storage cannot be allocated for the object.
`Undefined size for field TYPE.`	Compile Error	Member (field) declaration has no size for the declared type. You probably declared the member with an empty `struct` or `union` type declaration.
`Undefined size for field TYPE NAME.`	Compile Error	Member (field) declaration has no size for the declared type. You probably declared the member with an empty `struct` or `union` type declaration.
`Undefined size for parameter TYPE NAME.`	Compile Error	You declared a parameter with an incomplete type. Because the size of an incomplete type is unknown, storage cannot be allocated for the object.
`Undefined size for static TYPE NAME.`	Compile Error	You declared a `static` object with an incomplete type or without an initialization expression from which the compiler can calculate a size for the type. Because the size of an incomplete type is unknown, you cannot allocate storage for the object.
`Undefined static TYPE NAME.`	Compile Warning or Error	You declared the `static` function but never defined it. Because a `static` function is only visible within the file in which you declared it, you must define it at some point within the file in order to use it. If you called the function anywhere in the file, this diagnostic is an error. Otherwise it is a warning.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Undefined symbol NAME.`	Link Error	You used a variable or function in the project but did not define it anywhere.
`Unexpected #elif; #endif expected.`	Compile Error	Compiler encountered an `#else` preprocessor directive immediately prior to this `#elif` at the same level of conditional inclusion. Ensure that the conditional preprocessor `#include` directives at this level are in the proper order.
`Unexpected #elif; #if not seen.`	Compile Error	Compiler encountered an `#elif` preprocessor directive but it has not yet seen a beginning `#if`, `#ifdef`, or `ifndef` at this level.
`Unexpected #else; #endif expected.`	Compile Error	Compiler encountered an `#else` preprocessor directive immediately following a prior `#else` at the same level of conditional inclusion. Ensure that the conditional preprocessor `#include` directives at this level are in the proper order.
`Unexpected #else; #if not seen.`	Compile Error	Compiler encountered an `#else` preprocessor directive, but it has not yet seen a beginning `#if`, `#ifdef`, or `ifndef` at this level.
`Unexpected #endif; no matching #if, #ifdef, or #ifndef.`	Compile Error	Compiler encountered an `#endif` preprocessor directive but has not yet seen a beginning `#if, #ifdef`, or `ifndef` at this level.
`Unexpected EOF.`	Load Error	LabWindows/CVI encountered an unexpected End Of File (EOF) condition when loading an external object module. Ensure that the object file has not been truncated.
`Unexpected EOF; TOKEN expected.`	Compile Error	The compiler encountered an End Of File (EOF) condition while parsing a syntactic construct. Ensure that syntactic structure is complete, such as matching parenthesis and matching braces.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Unexpected end of format string.`	Non-Fatal Run-time Error	Format string you passed to the function is not complete. It is missing a source or destination format specifier, or contains an incomplete format specifier.
`Unexpected token.`	Compile Error	Compiler encountered an unexpected token while processing a `#define` preprocessor directive. Check for missing `)` in your macro parameter lists.
`Unexpected trailing tokens on directive line ignored.`	Compile Warning	Preprocessor line contains harmless trailing tokens that the compiler ignored.
`Uninitialized pointer.`	Non-Fatal Run-time Error	You never assigned a value to the pointer argument you passed to a function.
`Uninitialized pointer argument to library function.`	Fatal Run-time Error	Pointer expression you passed to the library function is invalid because you did not initialize it. It is either a local variable or an object in dynamic memory that you did not initialize.
`Uninitialized string.`	Non-Fatal Run-time Error	You never assigned a value to the pointer argument you passed to the library function, or it is `NULL`.
`'union NAME' declared inside parameter list has scope only for this declaration.`	Compile Warning	Union declared in the parameter list has scope only within the parameter list. As a result, its type is incompatible with all other types. You should declare the union type before you declare function types that use it.
`Unknown enumeration NAME.`	Compile Error	*`NAME`* is an undeclared enumeration type.
`Unknown field NAME of TYPE.`	Compile Error	Member selection or dereference has attempted to access an undeclared member, (field) name of a `struct`, or `union` type. Ensure that the member is declared for the `struct` or `union` type you select or dereference.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Unknown modifier.`	Non-Fatal Run-time Error	One of the modifiers in a format specifier is not valid.
`Unknown or unsupported OMF record at position NUMBER: OMF record type NUMBER.`	Load Error	LabWindows/CVI encountered an unknown OMF record while loading an external object module. Ensure that the external object module was compiled properly.
`Unknown size of type TYPE.`	Compile Error	You performed pointer arithmetic on operand(s) that are pointers to types of unknown size. The types are probably incomplete types or pointer to function types. Ensure that the pointer types point to fully declared types and are not pointers to functions.
`Unknown specifier.`	Non-Fatal Run-time Error	Specifier in the format specifier is not valid.
`Unnamed pop matching named push.`	Compile Warning	`pack` pragma used an unnamed pop that balances a name push.
`Unrecognized character escape sequence.`	Compile Warning	Character escape sequence does not conform to any known character escape sequence, octal escape sequence, or hexadecimal escape sequence.
`Unrecognized character escape sequence CHAR.`	Compile Warning	Character escape sequence does not conform to any known character escape sequence, octal escape sequence, or hexadecimal escape sequence.
`Unrecognized declaration.`	Compile Error	Declaration is unrecognizable. Check the declaration syntax for the function, object, or type you want to use.
`Unrecognized preprocessor directive.`	Compile Error	# character begins an unknown preprocessor directive. Check the spelling of the preprocessor directive.
`Unrecognized statement.`	Compile Error	Statement syntax is unrecognizable. Check the statement syntax for the type of statement you want to use.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Unsigned operand of unary –.`	Compile Warning	You performed a nonsensical unary negation operation on an unsigned type. A negation operation on an unsigned type is not effective.
`Unsupported segment combination type NUMBER: segment name NAME.`	Load Error	LabWindows/CVI encountered a bad segment while loading an external object module. Ensure that the external object module was compiled properly.

`Use of keyword '__import' contradicts previous use of keyword '__export' at POSITION.`	Compile Error	Use of a keyword in a variable definition contradicts a previous definition, for example:
		`int __export x;`
		`int __import x=0;`

`Use of keyword '__export' contradicts previous use of keyword '__import' at POSITION.`	Compile Error	Use of a keyword in a variable definition contradicts a previous definition, for example:
		`int __import x;`
		`int __export x=0;`

`Use of keyword '__declspec(dllimport)' contradicts previous use of keyword '__declspec(dllexport)' at POSITION.`	Compile Error	Use of a keyword in a variable definition contradicts a previous definition, for example:
		`int __declspec(dllexport) x;`
		`int __declspec(dllimport) x=0;`

`Use of keyword '__declspec(dllexport)' contradicts previous use of keyword '__declspec(dllimport)' at POSITION.`	Compile Error	Use of a keyword in a variable definition contradicts a previous definition, for example:
		`int __declspec(dllimport) x;`
		`int __declspec(dllexport) x=0;`

`Value parameter expected.`	Non-Fatal Run-time Error	You passed a pointer for a format specifier that requires a scalar value.

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Variables defined as DLL imports cannot be defined with an initial value.`	Compile Error	You assigned an initial value to a variable defined as a DLL import, for example: `int __import i = 0;` You must initialize the variable in a separate assignment statement.
`VXI address must be a multiple of 2 for word transfer.`	Fatal Run-time Error	You attempted to perform VXI word transfer beginning at an odd address.
`VXI address must be a multiple of 4 for longword transfer.`	Fatal Run-time Error	You attempted to perform a VXI longword transfer beginning at an address that is not a multiple of 4.
`w modifier not valid with l format specifier.`	Non-Fatal Run-time Error	You cannot use the `w` modifier with the `l` format specifier.
`Warning: Import libraries other than the one for the current compatibility mode may not work for symbols exported from an object file. It is recommended that you export using header files instead.`	DLL Link Warning	When creating a DLL using the Symbols Marked for Export method for specifying exported symbols, one of the modules was an object or library file. LabWindows/CVI does not have sufficient information to ensure that the import libraries it generates for all four compatible external compilers will have the correct names of the symbols in that module.
`WatchPoint: module name is not valid.`	Watchpoint Error	Module name you specified in the watch point is not present in the project or in any of the loaded instrument drivers.
`z modifier only valid if rep is present.`	Non-Fatal Run-time Error	`z` modifier cannot be used if the format specifier is not for an array.
`z modifier required to match string parameter.`	Non-Fatal Run-time Error	If you want to treat a character string as an array of another type, you must use the `z` modifier. This error also can occur if the order of the arguments is incorrect, or if an argument is missing.

B

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call 512 795 6990. You can access these services at:

United States: 512 794 5422
Up to 14,400 baud, 8 data bits, 1 stop bit, no parity

United Kingdom: 01635 551422
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

France: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as `anonymous` and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the `/support` directories.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name _____

Company _____

Address _____

_____

Fax (___) _____Phone (___) _____

Computer brand_____ Model _____Processor_____

Operating system (include version number) _____

Clock speed _____MHz RAM _____MB Display adapter _____

Mouse ___yes ___no Other adapters installed_____

Hard disk capacity _____MB Brand_____

Instruments used _____

_____

National Instruments hardware product model _____ Revision _____

Configuration _____

National Instruments software product _____ Version _____

Configuration _____

The problem is: _____

_____

_____

_____

_____

List any error messages: _____

_____

_____

The following steps reproduce the problem: _____

_____

_____

_____

_____

LabWindows/CVI Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

Hardware revision _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

National Instruments software _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

Operating system version _____

Operating system mode _____

Programming language _____

Programming language version _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: *LabWindows/CVI Programmer Reference Manual*

Edition Date: February 1998

Part Number: 320685D-01

Please comment on the completeness, clarity, and organization of the manual.

_____

_____

_____

_____

_____

_____

_____

If you find errors in the manual, please record the page numbers and describe the errors.

_____

_____

_____

_____

_____

_____

_____

Thank you for your help.

Name _____

Title _____

Company _____

Address _____

_____

E-Mail Address _____

Phone (___) _____ Fax (___) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 78730-5039

Fax to: Technical Publications
National Instruments Corporation
512 794 5678

Glossary

Prefix	Meaning	Value
m-	milli-	10^{-3}
µ-	micro-	10^{-6}
n-	nano-	10^{-9}

A

active window — The window user input affects at a given moment. The title of an active window is highlighted.

API — Application Programming Interface. A set of functions exported by a library.

Array Display — A mechanism for viewing and editing numeric arrays.

auto-exclusion — A mechanism that prevents pre-existing lines from executing in the Interactive Execution window.

B

binary control — A function panel control that resembles a physical on/off switch and can produce one of two values depending upon the position of the switch.

breakpoint — An interruption in the execution of a program.

Breakpoint — A function that interrupts the execution of a program.

Breakpoint command — A specific command that interrupts the execution of a program.

C

cdecl	A function calling convention in which function calls pass arguments from right to left, and the caller restores the stack position after the call.
check box	A dialog box item that allows you to toggle between two possible options.
clipboard	A temporary storage area LabWindows/CVI uses to hold text that is cut, copied, or deleted from a work area.
control	An input and output device that appears on a function panel for specifying function parameters and displaying function results.
cursor	The flashing rectangle that shows where you can enter text on the screen.
cursor location indicator	An element of the LabWindows/CVI screen that specifies the row and column position of the cursor in the window.

D

default command	The action that takes place when you press <Enter> and did not specifically select a command. A double outline indicates default command buttons in dialog boxes.
dialog box	A prompt mechanism in which you specify additional information necessary to complete a command.
DLL	Dynamic Link Library. A file that contains a collection of functions that multiple applications (`.exe` files) can use.

E

entry mode indicator	An element of the LabWindows/CVI screen that indicates the current text entry mode as either insert or overwrite.
excluded code	Code that LabWindows/CVI ignores during compilation and execution. LabWindows/CVI displays excluded lines of code in a different color than included lines of code.

F

`.fp` file	A file that contains information about the function tree and function panels of an instrument module.
function panel	A screen-oriented user interface to the LabWindows/CVI libraries in which you can interactively execute library functions and generate code for inclusion in a program.
Function Panel Editor window	The window in which you build a function panel. The *LabWindows/CVI Instrument Driver Developers Guide*.
function panel window	The window in which you can use function panels.
function tree	The hierarchical structure in which the functions in a library or an instrument driver are grouped. The function tree simplifies access to a library or instrument driver by presenting functions organized according to the operation they perform, as opposed to a single linear listing of all available functions.
Function Tree Editor window	The window in which you build the skeleton of a function panel file. It is described in the *LabWindows/CVI Instrument Driver Developers Guide*.

G

Generated Code box	A small box located at the bottom of the function panel screen that displays the function call code that corresponds to the current state of the function panel controls.
global control	A function panel control that displays the contents of global variables in a library function. Global controls allow you to monitor global variables in a function that the function does not specifically return as results by the function. These are read-only controls that cannot be altered by the user, and do not contribute a parameter to the generated code.
glue code	Special code that provides the interface between 32-bit LabWindows/CVI applications and 16-bit DLLs.

H

hex	Hexadecimal.

highlight — The way in which input focus is displayed on a LabWindows/CVI screen; to move the input focus onto an item.

I

immediate action command — A menu bar item that has no menu items associated with it and causes a command to execute immediately when you select it. An immediate action command is suffixed with an exclamation point (!).

input control — A function panel control that accepts a value you type in from the keyboard. An input control can have a default value associated with it. This value appears in the control when the panel is first displayed.

input focus — Displayed on the screen as a highlight on an item, signifying that the item is active. User input affects the item in the dialog box that has the input focus.

instrument driver — A set of high-level functions for controlling an instrument. It encapsulates many low-level operations, such as data formatting and GPIB, RS-232, and VXI communication, into intuitive, high-level functions. An instrument driver can pertain to one particular instrument or to a group of related instruments. An instrument driver consists of a program and a set of function panels. The program contains the code for the high-level functions. Associated with the instrument program is an include file that declares the high-level functions you can call, the global variables you can access, and the defined constants you can use.

Interactive Execution window — A LabWindows/CVI work area in which you can execute sections of code without creating an entire program.

L

list box	A dialog box item that displays a list of possible choices.
lvalue	A C expression that refers to an object that can be examined and modified. The name lvalue comes from the fact that only lvalues may appear on the left side of an assignment. Examples of lvalues are variables, parameters, array element references such as `a[i]`, `struct` element references such as `s->name` or `s.name`, and pointer dereferences such as `*ptr`. Expressions that are not lvalues are called rvalues.

M

MB	Megabytes of memory.
menu	An area accessible from the command bar that displays a subset of the possible command choices.

O

output control	A function panel control that displays a value that the function you execute generates. An output control parameter must be a string, an array, or a reference parameter of type integer, long, single-precision, or double-precision.
ordinal number	A numeric value that corresponds to a function within a DLL. The linker that creates the DLL arbitrarily defines it, or it may be specified in the `.def` file when the DLL is created.

P

PASCAL	A Windows 3.1 function calling convention in which function arguments are passed left to right, and the function restores the stack pointer before it returns.
Project window	A window that contains a list of files your application uses.
prompt command	A command that requires additional information before it can be executed; a prompt command appears on a pull-down menu suffixed with an ellipsis (. . .).

R

return value control	A function panel control that displays a value returned from a function as a return value rather than as a formal parameter.
ring control	A function panel control that represents a range of values much like the slide control, but displays only a single item in a list, rather than displaying the whole list at once as the slide control does. Each item has a different value associated with it. This value is placed in the function call.
rvalue	Any C expression that is not an lvalue. Examples of rvalues are array names, functions, function calls such as `f()`, assignment expressions such as `x = e` and cast expressions such as `(AnyType)e`.

S

SDK	Windows Software Development Kit. An API in the Windows operating system.
scroll bars	Areas along the bottom and right sides of a window that show your relative position in the file. You can use scroll bars can be used to move about in the window.
scrollable text box	A dialog box item that displays text in a scrollable display.
select	To choose the item that the next executed action affects by moving the input focus (highlight) to a particular item or area.
shortcut key command	A combination of keystrokes that provide a means of executing a command without accessing a menu in the command bar.
slide control	A function panel control that resembles a physical slide switch. A slide control is a means for selecting one item from a list of options; it inserts a value in a function call that depends upon the position of the cross-bar on the switch.
slider	The cross-bar on the slide control that determines the value placed in the function call.
Source window	A LabWindows/CVI work area where you edit and execute programs.

Standard Input/Output window	A LabWindows/CVI work area in which textual output to and input from the user take place.
standard libraries	The LabWindows/CVI User Interface, Analysis, Data Formatting and I/O, GPIB, GPIB-488.2, DDE, TCP, RS-232, Utility, and C system libraries.
stdcall	A Windows 95/NT calling convention in which function calls pass arguments from right to left, and the function restores the stack pointer before it returns.
String Display window	A window for viewing and editing string variables and arrays.

T

text box	A dialog box item in which you enter text from the keyboard or view text.

U

User Interface Editor window	The window in which you build pull-down menus, dialog boxes, panels, and controls and save them to a User Interface Resource (`.uir`) file. The *LabWindows/CVI User Interface Reference Manual* describes it.

V

Variables window	A window that shows the values of the variables that are currently active.

W

Watch window	A window that shows the values of selected variables and expressions that are currently active.
window	A working area that supports specific tasks related to developing and executing programs.

Index

Special Characters/Numbers

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X