LabWindows®/CVI

Programmer Reference Manual

July 1996 Edition

Part Number 320685C-01

© Copyright 1994, 1996 National Instruments Corporation.
All rights reserved.

Internet Support

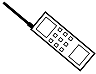
GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabVIEW: lv.support@natinst.com
LabWindows: lw.support@natinst.com
HiQ: hiq.support@natinst.com
Lookout: lookout.support@natinst.com
VISA: visa.support@natinst.com
FTP Site: ftp.natinst.com
Web Address: www.natinst.com

Bulletin Board Support

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

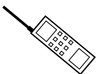
FaxBack Support

(512) 418-1111



Telephone Support (U.S.)

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



International Offices

Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 90 527 2321, France 1 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,
Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 95 800 010 0793,
Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085,
Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, U.K. 01635 523545

National Instruments Corporate Headquarters

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

Warranty

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

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

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

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

Copyright

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

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......ix
Organization of This Manual......ix
Conventions Used in This Manual......x
Related Documentation......xi
Customer Communication......xi

Chapter 1
LabWindows/CVI Compiler......1-1
Overview......1-1
LabWindows/CVI Compiler Specifics......1-1
Compiler Limits......1-1
Compiler Options......1-2
Compiler Defines......1-4
C Language Extensions......1-5
Keywords That Are Not ANSI C Standard......1-5
Calling Conventions (Windows 95 and NT Only)......1-6
Import and Export Qualifiers......1-6
C++-Style Comment Markers......1-7
Duplicate Typedefs......1-7
Structure Packing Pragma (Windows 3.1 and Windows 95/NT only)......1-7
Program Entry Points (Windows 95 and NT only)......1-8
C Library Issues......1-8
Using the Low-Level I/O Functions......1-8
C Data Types and 32-bit Compiler Issues......1-9
Data Types......1-9
Converting 16-bit Source Code to 32-bit Source Code......1-9
Debugging Levels......1-10
User Protection......1-11
Array Indexing and Pointer Protection Errors......1-11
Pointer Arithmetic (Non-Fatal)......1-12
Pointer Assignment (Non-Fatal)......1-12
Pointer Dereference Errors (Fatal)......1-13
Pointer Comparison (Non-Fatal)......1-13
Pointer Subtraction (Non-Fatal)......1-14
Pointer Casting (Non-Fatal)......1-14
Dynamic Memory Protection Errors......1-14
Memory Deallocation (Non-Fatal)......1-14
Memory Corruption (Fatal)......1-15
General Protection Errors......1-15
Library Protection Errors......1-15
Disabling User Protection......1-16
Disabling Protection Errors at Run-time......1-16

Disabling Library Errors at Run-time 1-16
Disabling Protection for Individual Pointer 1-16
Disabling Library Protection Errors for Functions 1-17
Details of User Protection 1-18
Pointer Casting 1-18
Dynamic memory 1-19
Library Functions 1-19
Unions 1-19
Stack Size 1-20
Include Paths 1-20
Include Path Search Precedence 1-20

Chapter 2
Using Loadable Compiled Modules 2-1
About Loadable Compiled Modules 2-1
Advantages and Disadvantages of Using Loadable Compiled Modules in LabWindows/CVI 2-1
Using a Loadable Compiled Module as an Instrument Driver Program File....2-2
Using a Loadable Compiled Module as a User Library 2-2
Using a Loadable Compiled Module in the Project List 2-3
Using a Loadable Compiled Module as an External Module 2-3
Special Considerations When Using a Loadable Compiled Module 2-4
Compiled Modules Using Asynchronous Callbacks 2-5

Chapter 3
Windows 95 and NT Compiler/Linker Issues 3-1
Loading 32-bit DLLs under Windows 95 and NT 3-1
DLLs for Instrument Drivers and User Libraries 3-2
Using The LoadExternalModule Function 3-2
Link Errors when Using DLL Import Libraries 3-2
DLL Path (.pth) Files Not Supported 3-2
16-Bit DLLs Not Supported 3-2
Generating an Import Library 3-3
Default Unloading/Reloading Policy 3-3
Compatibility with External Compilers 3-3
Choosing Your Compatible Compiler 3-4
Object Files, Library Files, and DLL Import Libraries 3-4
DLLs 3-4
Structure Packing 3-4
Bit Fields 3-5
Returning Floats and Doubles 3-5
Returning Structures 3-5
Enum Sizes 3-5
Long Doubles 3-6
Differences with the External Compilers 3-6
External Compiler Versions Supported 3-6

Required Preprocessor Definitions....3-6
Multithreading and the LabWindows/CVI Libraries....3-7
Multithread-Safe Libraries....3-7
Libraries that are Not Multithread Safe....3-7
Using LabWindows/CVI Libraries in External Compilers....3-8
Include Files for the ANSI C Library and the LabWindows/CVI Libraries....3-9
Standard Input/Output Window....3-9
Resolving Callback References From .UIR Files....3-9
Linking to Callback Functions Not Exported From a DLL....3-10
Resolving References from Modules Loaded at Run-Time....3-11
Resolving References to Non-LabWindows/CVI Symbols....3-11
Resolving Run-Time Module References to Symbols Not Exported From a DLL....3-12
Run State Change Callbacks Are Not Available in External Compilers....3-12
Calling InitCVIRTE and CloseCVIRTE....3-13
Creating Object and Library Files in External Compilers for Use in LabWindows/CVI....3-14
Microsoft Visual C/C++....3-14
Borland C/C++ command line compiler....3-14
WATCOM C/C++....3-15
Symantec C/C++....3-15
Creating Executables in LabWindows/CVI....3-16
Creating DLLs in LabWindows/CVI....3-16
Customizing an Import Library....3-16
Preparing Source Code for Use in a DLL....3-17
Calling Convention for Exported Functions....3-17
Exporting DLL Functions and Variables....3-18
Include File Method....3-18
Export Qualifier Method....3-18
Marking Imported Symbols in Include File Distributed with DLL....3-19
Recommendations....3-20
Automatic Inclusion of Type Library Resource for Visual Basic....3-20
Creating Static Libraries in LabWindows/CVI....3-21
Creating Object Files in LabWindows/CVI....3-21
Calling Windows SDK Functions in LabWindows/CVI....3-22
Windows SDK Include Files....3-22
Using Windows SDK Functions for User Interface Capabilities....3-22
Using Windows SDK Functions to Create Multiple Threads....3-23
Automatic Loading of SDK Import Libraries....3-23
Setting Up Include Paths for LabWindows/CVI, ANSI C, and SDK Libraries....3-23
Compiling in LabWindows/CVI for Linking in LabWindows/CVI....3-24
Compiling in LabWindows/CVI for Linking in an External Compiler....3-24
Compiling in an External Compiler for Linking in an External Compiler....3-24
Compiling in an External Compiler for Linking in LabWindows/CVI....3-24
Handling Hardware Interrupts under Windows 95 and NT....3-24

Chapter 4
Windows 3.1 Compiler/Linker Issues..4-1
Using Modules Compiled by LabWindows/CVI..4-1
Using 32-Bit Watcom Compiled Modules Under Windows 3.1 4-1
Using 32-Bit Borland or Symantec Compiled Modules Under Windows 3.1............4-2
16-Bit Windows DLLs..4-4
Helpful LabWindows/CVI Options for Working with DLLs 4-4
DLL Rules and Restrictions 4-5
DLL Glue Code..4-7
DLLs That Can Use Glue Code Generated at Load Time 4-8
DLLs That Cannot Use Glue Code Generated at Load Time 4-8
Loading a DLL That Cannot Use Glue Code Generated at Load Time 4-8
Rules for the DLL Include File Used to Generate Glue Code .4-9
If the DLL Requires a Support Module Outside of the DLL ...4-9
If the DLL is Passed Arrays Bigger Than 64 K 4-10
If the DLL Retains a Buffer After the Function Returns (an Asynchronous Function) 4-11
If the DLL Calls Directly Back Into 32-Bit Code...................4-12
If the DLL returns pointers..4-14
If a DLL Is Passed a Pointer That Points to Other Pointers.....4-16
DLL Exports Functions by Ordinal Value Only 4-18
Generated Glue Code: 4-19
Recognizing Windows Messages Passed from a DLL..................................4-19
RegisterWinMsgCallback 4-19
UnRegisterWinMsgCallback 4-20
GetCVIWindowHandle..4-20
Creating 16-bit DLLs with Microsoft Visual C++ 1.5..................................4-21
Creating 16-bit DLLs with Borland C++ 4-21
DLL Search Precedence 4-22

Chapter 5
UNIX Compiler/Linker Issues 5-1
Calling Sun C Library Functions 5-1
Restrictions on Calling Sun C Library Functions 5-1
Creating Executables...5-1
Run State Change Callbacks Are Not Available in Executables 5-2
Main Function Must Call InitCVIRTE 5-2
Using Externally Compiled Modules..5-3
Restrictions on Externally Compiled Modules 5-3
Compiling Modules With External Compilers 5-3
Locking Process Segments into Memory Using plock()...5-4

Chapter 6
Building Multiplatform Applications

Building Multiplatform Applications 6-1
Multiplatform Programming Guidelines 6-1
Library Issues 6-1
Externally Compiled Module Issues 6-3
Multiplatform User Interface Guidelines 6-3

Chapter 7
Creating and Distributing Standalone Executables and DLLs

Creating and Distributing Standalone Executables and DLLs 7-1
Introduction to the Run-Time Engine 7-1
Distributing Standalone Executables under Windows 7-1
Minimum System Requirements for Windows 95 and NT 7-1
No Math Coprocessor Required for Windows 95 and NT 7-2
Minimum System Requirements for Windows 3.1 7-2
Math Coprocessor Software Emulation for Windows 3.1 7-2
Distributing Standalone Executables under UNIX 7-2
Distributing Standalone Executables under Solaris 2 7-3
Distributing Standalone Executables under Solaris 1 7-4
Minimum System Requirements for UNIX 7-5
Configuring the Run-Time Engine 7-5
Translating the Message File 7-5
Option Descriptions 7-6
cvirt*x* (Windows 3.1 Only) 7-6
cvidir 7-6
Necessary Files for Running Executable Programs 7-7
Necessary Files for Using DLLs Created in Windows 95/NT 7-8
Location of Files on the Target Machine for Running Executables and DLLs 7-8
LabWindows/CVI Run-Time Engine on Windows 95/NT 7-9
LabWindows/CVI Run-Time Engine on Windows 3.1 7-9
Rules for Accessing UIR, Image, and Panel State Files on All Platforms 7-9
Rules for Using DLL Files under Windows 95 and NT 7-10
Rules for Using DLL Files in Windows 3.1 7-10
Rules for Loading Files Using LoadExternalModule 7-11
Forcing Modules Referenced by External Modules into Your Executable or DLL 7-12
Using LoadExternalModule on Files in the Project 7-12
Using LoadExternalModule on Library and Object Files Not in the Project 7-13
Using LoadExternalModule on DLL Files under Windows 95 and NT 7-14
Using LoadExternalModule on DLL and Path Files under Windows 3.1 7-14
Using LoadExternalModule on Source Files (.c) 7-15
Rules for Accessing Other Files 7-16
Error Checking in your Standalone Executable or DLL 7-16

Chapter 8
Distributing Libraries and Function Panels 8-1
How to Distribute Libraries 8-1
Adding Libraries to User's Library Menu 8-1
Specifying Library Dependencies 8-2

Appendix A
Errors and Warnings A-1

Appendix B
Error Checking in LabWindows/CVI B-1
Error Checking B-2
Status Reporting by LabWindows/CVI Libraries and Instrument Drivers B-3
User Interface Library B-3
Analysis/Advanced Analysis Libraries B-3
Data Acquisition Library B-3
VXI Library B-4
GPIB/GPIB 488.2 Library B-4
RS-232 Library B-4
TCP Library B-5
DDE Library B-5
X Property Library B-5
Formatting and I/O Library B-5
Utility Library B-6
ANSI C Library B-6
LabWindows/CVI Instrument Drivers B-6

Appendix C
Customer Communiation C-1

Glossary G-1

Index I-1

Figures

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

Tables

Table 1-1. LabWindows/CVI Compiler Limits 1-1
Table 1-2. LabWindows/CVI Allowable Data Types 1-9
Table 1-3. Stack Size Ranges for LabWindows/CVI 1-20

Table A-1. Error Messages A-1

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, and Windows fundamentals.

Organization of This Manual

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

- Chapter 1, *LabWindows/CVI Compiler*, describes LabWindows/CVI compiler specifics, 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 different kinds of compiled modules available under LabWindows/CVI and includes programming guidelines for modules generated by external compilers.
- Chapter 3, *Building Multiplatform Applications*, contains guidelines and caveats for writing platform-independent LabWindows/CVI applications. LabWindows/CVI currently runs under Windows for the PC and Solaris for the SPARCstation.
- Chapter 4, *Creating and Distributing Standalone Executables,* describes how the Run-Time Engine, DLLs, external compiled modules, and other files interact with your executable file. This chapter also describes the technique of error checking in a standalone executable program. You can create executable programs from any project that runs in the LabWindows/CVI environment.
- Chapter 5, *Distributing Libraries*, describes how to distribute libraries, how to add libraries to the **Library** menu, and how to specify library dependencies.
- 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, *Error Checking in LabWindows/CVI*, describes error checking codes in the LabWindows/CVI environment and how errors are reported in LabWindows/CVI libraries and instruments.

- Appendix C, *Customer Communication*, contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation.
- 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.

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

Acronyms, abbreviations, metric prefixes, mnemonics, and symbols, and terms are listed in the *Glossary*.

Related Documentation

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

- *Microsoft Windows 3.1 Programmer's Reference Manual*, volumes 1 and 2, Microsoft Corporation, Redmond WA, 1987–1992.
- *WATCOM C/386 User's Guide*, WATCOM Publications Limited, Waterloo, Ontario, Canada, 1992.
- 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 C, *Customer Communication*, at the end of this manual.

Chapter 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 focuses instead 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

The compiler limits for LabWindows/CVI are shown in Table 1-1.

Table 1-1. LabWindows/CVI Compiler Limits

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 with the **Compiler Preferences** command in the **Options** menu of a Project window. This command invokes a dialog box which allows you to set the following LabWindows/CVI compiler options.

- **Compatibility with** (Windows 95/NT only) displays the current compiler compatibility mode. (For more information on external compiler compatibility, see the *Compatibility with External Compilers* section in Chapter 3, *Windows 95 and NT Compiler/Linker Issues*.)

- **Default calling convention** (Windows 95/NT only) sets the compiler's default calling convention, unless the compatible compiler is WATCOM. For the other compilers, the default calling convention is normally `cdecl` but can be changed to `stdcall`. For WATCOM, it is always the stack-based calling convention. Do not change the default calling convention to `stdcall` if you plan to generate static library or object files for all four compatible external compilers. (For more information, see the *Calling Conventions (Windows 95/NT Only)* section in Chapter 3, *Windows 95 and NT Compiler/Linker Issues*.)

- **Maximum number of compile errors** sets an upper limit on the number of compiler errors to be listed in the Build Errors window.

- **Require function prototypes** requires all function references to be preceded by a full prototype declaration. A full prototype is one that includes the function return type as well as the types of each parameter. If a function has no parameters, a full prototype must have the `void` keyword to indicate this case. A *new style* function definition (one in which parameters are declared directly in the parameter list) serves as a prototype.

  Missing prototype errors can occur at the following places:

  – Typedefs such as `typedef void FUNTYPE()`

  – Function pointer declarations such as `void (*fp)()` whether used as a global, local, parameter, array element or structure member

  – *Old style* function definitions (one in which parameters are declared outside of the parameter list) that are not preceded by a full prototype

  – Function call expressions such as `(*fp)()`, where `fp` does not have a full prototype

  Caution: ***Be sure you enable the*** **Require Function Prototypes** ***option. If disabled, some of the run-time error checking will also be disabled.***

- **Require return values for non-void functions** generates compile warnings for non-void functions (except `main`) that do not end with a `return` statement returning a value. LabWindows/CVI reports a run-time error when a non-void function executes without returning a value.

For example, the following code produces a compile-time warning and will produce a run-time error when `flag` is FALSE.

```
int fun (void)
{
   if (flag) {
      return 0;
   }
}
```

- **Enable signed/unsigned pointer mismatch warning** generates a compiler warning for pointer assignments in which the left side and right side are not both `signed` or `unsigned` expressions. According to the ANSI C standard, these assignments should be errors because they involve incompatible types. In practice, however, assigning a pointer to `unsigned` *type*, the value of a pointer to `signed` *type*, or vice versa, causes no problems.

  The LabWindows/CVI compiler checks assignment statements and function call arguments to ensure that the lvalue and rvalue expressions have compatible types. If you select **Enable signed/unsigned pointer mismatch warning**, LabWindows/CVI generates compile warnings when the lvalue and rvalue expressions are both pointers to integers but when one points to a signed integer and the other points to an unsigned integer. For example, the LabWindows/CVI compiler would generate a `signed type mismatch between pointer to char and pointer to unsigned char` warning on the call to `MyFunction` in the following code example.

```
void MyFunction (unsigned char *x);
char *y = "my string";
main () {
         MyFunction (y);
}
```

- **Enable unreachable code warning** generates a compiler warning for statements that cannot be reached on execution. When you select **Enable unreachable code warning**, the LabWindows/CVI compiler generates a warning at each line of code that cannot be reached during execution of your program. For example, a warning would be reported on the break statement in the following code.

```
switch (intval)   {
   case 4:
      return 0;
      break;
}
```

- **Track include file dependencies** keeps the project up to date by tracking the dependencies between source files and include files. Whenever a file is modified, LabWindows/CVI marks for compilation all project source files that include the modified file.

- **Prompt for include file paths** sets LabWindows/CVI to prompt you to make a manual search for any header files listed in the `#include` lines that the compiler cannot find.

When you find them, you can automatically insert the appropriate path into the Include Paths list for the project.

- **Stop on first file with errors** sets the LabWindows/CVI compiler to terminate compilation after one file is found to have errors. With this option, you can correct build errors in your project one file at a time.
- **Show Build Error Window for Warnings** sets the LabWindows/CVI compiler to bring up the Build Error window when warnings occur, even if there are no errors. If it is deactivated, warnings can occur without being brought to your attention.
- **Display status dialog during build** displays a status box during the build, showing the name of the file being compiled, the number of errors and warnings encountered, and a percent completed value. Your project compiles faster when you disable this feature.

Compiler Defines

The LabWindows/CVI compiler accepts compiler defines through the **Compiler Defines** command in the **Build** menu of the Project window.

Compiler defines have the syntax

`/Dx` or `/Dx=y`

where `x` is a valid C identifier. `x` can be used by the `#if` and `#ifdef` preprocessor directives for conditional compilation or as a predefined macro in your source code. If `y` contains embedded blanks, it must be surrounded by double quotation marks.

LabWindows/CVI predefines these macros to help you write platform-dependent code.

- `_CVI_` is defined to be 1 in version 3.0, `301` in version 3.0.1, and `310` in version 3.1.
- `_NI_mswin_` is defined if you compile under Windows 3.x, Windows 95, or Windows NT.
- `_NI_mswin16_` is defined if you compile under Windows 3.x.
- `_NI_mswin32_` is defined if you compile under Windows 95 and Windows NT.
- `_NI_i386_` is defined if you compile on a PC.
- `_NI_unix_` is defined if you compile with UNIX.
- `_NI_sparc_` is defined if you compile on a SPARCstation. The value of `_NI_sparc_` is `1` if you are running under Solaris 1.x and `2` if you are running under Solaris 2.x.
- `_CVI_DEBUG_` is defined as `1` only if the debugging level is other than "none."

The following predefined macros are defined for Windows 95 and NT.

- `_CVI_EXE_` is defined if the project target type is Standalone Executable

- `_CVI_DLL_` is defined if target type is Dynamic Link Library
- `_CVI_LIB_` is defined if target type is Static Library
- `__DEFALIGN` is defined to the default structure alignment (8 for Microsoft and Symantec, 1 for Borland and WATCOM)
- `_NI_VC_` is defined to `220` for the Microsoft Visual C/C++ compatibility mode
- `_NI_SC_` is defined to `720` for the Symantec C/C++ compatibility mode
- `_NI_BC_` is defined to `451` for the Borland C/C++ mode
- `_NI_WC_` is defined to `1050` for the WATCOM C/C++ mode
- `_WINDOWS` is defined
- `WIN32` is defined
- `_WIN32` is defined
- `__WIN32__` is defined
- `__NT__` defined
- `_M_IX86` is defined to 400
- `_NI_mswin32_` is defined
- `__FLAT__` is defined to 1

C Language Extensions

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

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.

Calling Conventions (Windows 95 and NT Only)

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

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

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

In WATCOM C/C++, the default calling convention is 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. (See the *Creating Object and Library Files in External Compilers for Use in LabWindows/CVI* section in Chapter 3, *Windows 95 and NT Compiler/Linker Issues*.) The `-4s` option causes the stack-based calling convention to be the default. In LabWindows/CVI in WATCOM compatibility mode, the default calling convention is always the stack-based convention. It cannot be changed. LabWindows/CVI does compile with the `cdecl` and `stdcall` conventions under WATCOM, except that floating point and structure return values do not work in the `cdecl` calling convention. It is recommended that you avoid using `cdecl` with WATCOM.

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

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

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

Import and Export Qualifiers

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

```
__declspec(dllimport)
__declspec(dllexport)
```

```
__import
__export
_import
_export
```

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

```
DLLIMPORT
DLLEXPORT
```

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

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

C++-Style Comment Markers

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

Duplicate Typedefs

The LabWindows/CVI compiler 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 structure can start on any 1-byte boundary, and there are no gaps between the structure elements.

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

You can set the maximum alignment as follows:

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

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

Program Entry Points (Windows 95 and NT only)

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

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

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. See the *UNIX Compiler/Linker Issues* chapter for more information.

Under Windows, you can use these functions if you include `lowlvlio.h`. No function panels are provided.

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

The LabWindows/CVI data types are shown in Table 1-2.

Table 1-2. LabWindows/CVI Allowable Data Types

Type	Size	Minimum	Maximum
`char`	8	-128	127
`unsigned char`	8	0	255
`short`	16	-32768	32767
`unsigned short`	16	0	65535
`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 are converting LabWindows for DOS applications to LabWindows/CVI applications, 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 there is little difference 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 two-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 four-byte integer. In most cases, this does not cause a problem and the conversion is transparent, because functions that used two-byte integers in LabWindows for DOS use four-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, two 16-bit values are packed into each element of the 32-bit array. Any attempt to access the array on an element-by-element basis will not work. Declare the array as `short` instead, and make sure any type specifiers that refer to it have the `[b2]` modifier when passed as an argument to a Formatting and I/O Library function.

- Uses an `int` variable in a way that requires it to be a two-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 will 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 will 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 need to make any modifications if the specifier for a variable of type*** `int` ***is*** `%d`***.***

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. LabWindows/CVI for Windows 3.1 runs 16-bit DLLs through a special interface generated from the header file for the DLL. See 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*, for more information.

Debugging Levels

You can compile the source modules in your application with debugging information so that you can use breakpoints and view or modify variables and expressions while your program is running. You set the debugging level by selecting the **Run Options** command in the **Options** menu of the Project window. The user selectable debugging levels are as follows:

- **None**—Source modules execute faster without debugging, but you sacrifice the ability to set breakpoints or to use the Variable Display window, and there is no user protection to check for bad pointers, over-indexing arrays, invalid array sizes, and so on.
- **Standard**—In this mode you can set breakpoints, use the Variable Display window, and you have user protection.
- **Extended**—In this mode, you have the same benefits of Standard mode along with added user protection that validates every attempt to free dynamically allocated memory by verifying that the address passed is actually the beginning of an allocated block.

User Protection

User protection detects invalid program behavior which could not otherwise have been determined 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.

There are two groups of user protection errors 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 described below.

- *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 may continue.
- *Fatal* errors include expressions that cannot be executed without causing major problems, such as aborting the program. 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 is further divided into subgroups as described in the following sections.

Array Indexing and Pointer Protection Errors

The pointer protection errors catch invalid operations with pointers and arrays. These errors are grouped by the type of expression causing 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 would result 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 resulted 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 pointer variables are assigned an invalid pointer value. 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 (assigned to a pointer 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 used as a function
- Dereference of function pointer used as data
- Dereference of an *n*-byte object where only *m* exist
- 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 a pointer expression is cast to type (`AnyType *`) and there is not enough space for an object of type `AnyType` at the location given by the expression. This occurs only when casting a dynamically allocated object for the first time, such as with the code `(double *) malloc(1)`, which returns the following error.

- Not enough space for casting expression to *`type`*

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`
- Cannot free: memory not allocated by `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 checks to verify the integrity of the memory blocks. When you set the Debugging Level to Extended, LabWindows/CVI thoroughly checks dynamic memory for each memory operation. LabWindows/CVI generates the following error when a problem is discovered.

- Dynamic memory is corrupt

General Protection Errors

LabWindows/CVI also checks for stack overflow (a fatal error) and for missing return values (a non-fatal error).

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

The missing return value error means that a non-void function (one that was not declared with `void` return type) has returned, but has 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
- Pointer to free memory passed to library function
- Array argument too small
- Scalar argument to library function; expected array
- Missing terminating null in string argument
- Argument must be a character

LabWindows/CVI reports other library dependent errors because the library function could not perform its task for some reason. These errors typically return a special value from the library function or set a global variable that indicates the error. However, if you have enabled **Break on Library Errors** in the **Run Options** command in the **Options** menu of the Project window,

LabWindows/CVI suspends execution after a library function reports one of these errors. A message appears that displays the name of the function and either the return value or a string explaining why the function failed.

Disabling User Protection

Occasionally, you may 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 programatically control whether LabWindows/CVI suspends execution when it encounters a general protection or library protection error. This function does not affect the **Break on Library Errors** feature.

Disabling Library Errors at Run-time

The **Break on Library Errors** checkbox in the **Run Options** command in the **Options** menu of the Project window defines the initial state for reporting library errors when executing the project. Additionally, you can use the `SetBreakOnLibraryErrors` function in the Utility Library to programatically control whether LabWindows/CVI suspends execution when a library function returns an error. Use of this function does not affect the reporting of general protection or 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 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 re-size it in an object module. The following steps describe how this error occurs.

1. You declare a pointer in a source module compiled with debugging enabled. You then assign to the pointer an address returned by `malloc` or `calloc`:

   ```
   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. You can do this with the function `realloc` or by freeing the pointer and then reassigning it to memory allocated by `malloc`:

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

   or

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

3. You use the same pointer in a source module compiled 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 shown above to disable checking for the pointer before it is used in the source module:

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

Disabling Library Protection Errors for Functions

You can also disable or enable library protection errors by placing pragmas in the source code. These pragmas are ignored 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. Nested include files are not affected.

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

The following two 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 and enable only the library errors for particular calls to a library function, without affecting the run-time checking of argument values, use the Utility Library function*** `SetBreakOnLibraryErrors`**.**

You cannot use pragmas to disable protection for the functions in the statically linked libraries (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. You can disable library protection for these functions at run-time, however, by using the Utility Library function*** `SetBreakOnLibraryErrors`***. See the*** *LabWindows/CVI Standard Libraries Reference Manual* ***for more information on the Utility Library.***

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 operand value to be converted to the type named within the parenthesis.

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

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

Note: ***There is an exception. The cast applied implicitly or explicitly to*** `void *` ***values obtained from*** `malloc` ***or*** `calloc` ***does not disable user protection.***

Casting from 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 also 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 which 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 may disable checking on the dynamic data, as explained in the *Pointer Casting* discussion in this section.

You can use the `realloc` function to re-size dynamically allocated memory. This function increases or decreases the size of the object associated with the dynamic memory.

Library Functions

The LabWindows/CVI library functions that have 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. Do not use a `void *` cast when passing an argument to a function that expects a variably typed 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

The stack is used for passing function parameters and storing automatic local variables. The maximum stack size can be set when you select the **Run Options** command in the **Options** menu of the Project window. LabWindows/CVI supports the following ranges.

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

Platform	Minimum	Default	Maximum
Windows 3.1	4 KB	16 KB	16 KB
Windows 95 and NT	100 KB	100 KB	1 MB
Solaris 1 for Sun	100 KB	100 KB	5 MB
Solaris 2 for Sun	100 KB	100 KB	5 MB

Note: ***For LabWindows/CVI for Windows 3.1, the actual stack size approaches 64 KB when the Debugging Level is 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 include files for the drivers are placed 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.

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

Chapter 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. See Chapter 3, *Windows 95 and 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 generated by external compilers.

About Loadable Compiled Modules

There are several ways to use 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 that they are accessible to any project. You can list compiled modules in your project, so that they are accessible only within that project. You can use compiled modules dynamically in your program with the functions, `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, containing one or multiple object modules
- A `.lib` file on the PC, or an `.a` file under UNIX, containing one or more object modules
- In Windows only, a `.dll` file containing a Windows dynamic-link library (DLL)

You can create any of these compiled modules in LabWindows/CVI under Windows 95 and NT, or using a compatible external compiler. On Windows 3.1, LabWindows/CVI can create `.obj` files only. Under UNIX, LabWindows/CVI or compatible compilers can create `.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 generated by LabWindows/CVI run faster than source modules because all of the debugging and user protection information has been removed. Compiled modules generated by external compilers can run faster because of optimization.

- Users of your program cannot modify your source code.

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

- You cannot debug compiled modules. To debug a program, you must set breakpoints and view the variables used by your program. Because compiled modules do not contain any debugging information, you cannot use these debugging features in those modules.
- 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 can also be a software utility. The use of instrument drivers is described in the *Using Instrument Drivers* and *Instruments Menu* sections of Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual*.

During debugging, load the instrument driver program file into LabWindows/CVI as a source file. After you debug it, you can compile the instrument driver program file. You can call an instrument driver from any code module in any project, and from the Interactive Execution window. If you enable **Require function prototypes** through the **Compiler Preferences** command in the **Options** menu, you must include the `.h` file for the instrument driver in the calling code module. You can manually load or unload instrument drivers at any time.

See 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 and NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, Chapter 5, *UNIX Compiler/Linker Issues*.

Using a Loadable Compiled Module as a User Library

You may install your own libraries into the **Library** menu. A user library has the same form as an instrument driver. Anything that can be loaded with the **Instruments** menu can be loaded as a user library, provided the program is in compiled form. See the *Using Instrument Drivers* and the *Instruments Menu* sections of Chapter 3, *Project Window*, of the *LabWindows/CVI User Manual* for more information. The main difference between modules loaded as instrument drivers and those loaded as user libraries is that instrument drivers can be unloaded using the **Unload** command in the **Instrument** menu, but user libraries cannot be unloaded. Also, because user libraries must be compiled, they cannot be edited while they are loaded as libraries.

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. In such a case, function panels may not be necessary for the library module. If the `.fp` file for the library module contains no classes or functions, the name does *not* appear in the **Library** menu when the user library is loaded.

You can call a compiled module loaded as a user library from any code module in any project and from the Interactive Execution window. If you enable **Require function prototypes** with the **Compiler Preferences** command in the **Options** menu, you must include the `.h` file for the user library in the module that is making the call.

See the *LabWindows/CVI Instrument Driver Developers Guide* for information about creating instrument drivers. You can create user libraries the same way you create instrument drivers.

User libraries must adhere to the requirements outlined for the target operating system. The operating system requirements are discussed in the following chapters: Chapter 3, *Windows 95 and NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, Chapter 5, *UNIX Compiler/Linker Issues*.

Using a Loadable Compiled Module in the Project List

You can call a compiled module in the project list from any code module in that project. If you enable **Require function prototypes** by selecting the **Compiler Preferences** command in the **Options** menu, you must include the `.h` file for the compiled module in the calling code module.

Compiled modules must adhere to the requirements outlined for the target operating system. The operating system requirements are discussed in the following chapters: Chapter 3, *Windows 95 and NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, Chapter 5, *UNIX Compiler/Linker Issues*.

Using a Loadable Compiled Module as an External Module

You can develop a special program and load it as an *external module*. You can load, execute, and unload this external module programmatically from a Source window or the Interactive Execution window, using the `LoadExternalModule`, `RunExternalModule`, and `UnloadExternalModule` functions. See Chapter 8, *Utility Library*, of the *LabWindows/CVI Standard Libraries Reference Manual* for more information on using these functions.

During debugging, list the external module in the project as a source file. After you debug it, you can compile the external module. External modules must adhere to the requirements outlined for the target operating system. The operating system requirements are discussed in the following chapters: Chapter 3, *Windows 95 and NT Compiler/Linker Issues*, Chapter 4, *Windows 3.1 Compiler/Linker Issues*, Chapter 5, *UNIX Compiler/Linker Issues*.

Special Considerations When Using a Loadable Compiled Module

You may need to notify certain compiled modules when the program starts, suspends, continues, or stops. For example, a compiled module that has asynchronous callbacks must not call the program callbacks when program execution is suspended at a breakpoint. LabWindows/CVI has a callback mechanism you can use to inform a compiled module of changes in the program status.

If you have to notify a compiled module of changes in the run state, install the callback automatically by naming it `__RunStateChangeCallback`, and including it in the compiled module. This callback must be in a compiled file, not in a source file. More than one compiled module may contain functions with this name, because it is never entered into the global name space. The prototype for the callback is as follows.

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

The actions are defined in `libsupp.h` as the following enumerated type:

```
enum {
  kRunState_Start,
  kRunState_Suspend,
  kRunState_Resume,
  kRunState_AbortingExecution,
  kRunState_Stop,
  kRunState_EnableCallbacks,
  kRunState_DisableCallbacks
};
```

Two examples of typical program state changes are listed below.

<u>Example 1</u>

```
   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 is suspended;  CVI environment resumes */
   .
   .
   .
```

```
    /* user requests the execution be resumed, via 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 is suspended;  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*** `Suspend` ***notification is not always followed by a*** `Resume` ***notification. A*** `Stop` ***notification can follow a*** `Suspend` ***notification without an intervening*** `Resume` ***notification.***

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

Compiled Modules Using Asynchronous Callbacks

If a compiled module calls a program function asynchronously (such as through interrupts or signals), it must announce the callback by calling `EnterAsyncCallback` before calling the callback, and calling `ExitAsyncCallback` after calling the callback. `EnterAsyncCallback` and `ExitAsyncCallback` have one parameter, which is a pointer to a buffer of size `ASYNC_CALLBACK_ENV_SIZE`. The same buffer must be passed into `ExitAsyncCallback` that was passed into `EnterAsyncCallback` because the buffer is used to store state information. The definition of `ASYNC_CALLBACK_ENV_SIZE` and the prototypes for these two functions are in `libsupp.h`.

Chapter 3
Windows 95 and NT Compiler/Linker Issues

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

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

- Load 32-bit DLLs, via the standard import library mechanism
- Create 32-bit DLLs and DLL import libraries
- Create library files as well as object files
- Call the LabWindows/CVI libraries from executables or DLLs created in any of the four compatible external compilers
- Create object files, library files, and DLL import libraries that can be used in the compatible external compilers
- Load object files, library files, and DLL import libraries created in any of the compatible external compilers.
- Call Windows SDK functions

This chapter discusses these capabilities.

Loading 32-bit DLLs under Windows 95 and NT

Under Windows 95 and NT, LabWindows/CVI can load 32-bit DLLs. Under Windows 3.1, LabWindows/CVI can load 16-bit DLLs only. Because the environment is 32-bit, special glue code is no longer needed under Windows 95 and NT. LabWindows/CVI links to DLLs via the standard 32-bit DLL import libraries that you generate when you create 32-bit DLLs with any of the compilers. Because DLLs are linked in this way, you can no longer specify a DLL file directly in your project. You must specify the DLL import library file instead.

DLLs for Instrument Drivers and User Libraries

Under Windows 95 and NT, a DLL is never directly associated with an instrument driver or user library. Instead, an instrument driver or user library can be associated with a DLL import library. Each DLL must have a DLL import library (`.lib`) file. In general, if the program for an instrument driver or user library is contained in a DLL, there must be a DLL import library in the same directory as the function panel (`.fp`) file. The DLL import library specifies the name of the DLL, which is then searched for using the standard Windows DLL search algorithm.

An exception is made to facilitate using VXI*plug&play* instrument driver DLLs. When you install a VXI*plug&play* instrument driver, the DLL import library is not placed in the same directory as the `.fp` file. If a `.fp` file is in the VXI*plug&play* directory, LabWindows/CVI searches for an import library in the VXI*plug&play* import library directory before it looks for a program file in the directory of the `.fp` file, unless a program file in the directory of the `.fp` file (and with the same base name as the `.fp` file) is listed in the project and is not excluded.

Using The LoadExternalModule Function

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

Link Errors when Using DLL Import Libraries

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

DLL Path (.pth) Files Not Supported

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

16-Bit DLLs Not Supported

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

Generating an Import Library

If you do not have a DLL import library or if the one you have contains references not exported by the DLL, you can generate an import library in LabWindows/CVI. You must have an include file that contains the declarations of all of the functions and global variables you want to access from the DLL. Load the include file into a Source window, and select the **Generate DLL Import Library** command in the **Options** menu.

Default Unloading/Reloading Policy

Some fundamental differences exist in the way DLLs being used by multiple processes are handled in Windows 95/NT and Windows 3.1.

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

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

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

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

Compatibility with External Compilers

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

Choosing Your Compatible Compiler

When installing LabWindows/CVI, you must choose your compatible compiler. If sometime later you want to change your choice of compatible compiler, you can run the installation program and 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, the file can be used only in the selected compatible compiler or in a copy of LabWindows/CVI that has been installed with the same compatibility choice.

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

DLLs

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

Structure Packing

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

If your DLL API uses structures, you should guarantee compatibility among the different compilers by using the `pack` pragma to specify a specific maximum alignment factor. You should place this pragma in the DLL include file, before the definitions of the structures. (You can choose any alignment factor.) After the structure definitions, you should 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 */
```

The __DEFALIGN predefined macro is defined to the default structure alignment.

Bit Fields

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

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

Returning Floats and Doubles

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

Returning Structures

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

Enum Sizes

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

Long Doubles

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

Differences with the External Compilers

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

External Compiler Versions Supported

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

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

Required Preprocessor Definitions

When 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 the LabWindows/CVI Libraries are called from an multithreaded executable or DLL created in an external compiler.
- When the LabWindows/CVI Libraries are called from a DLL created in LabWindows/CVI but loaded from a multithreaded executable created in another compiler.

Some, but not all, of the LabWindows/CVI libraries are multithread safe, as discussed in the following sections.

Multithread-Safe Libraries

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

- Analysis and Advanced Analysis
- GPIB (if you are using a native 32-bit driver)
- VXI
- VTL
- RS-232
- Data Acquisition
- Easy I/O for DAQ
- Formatting and I/O (except for the Standard I/O Window)
- ANSI C (except for the Standard I/O Window)

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

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

Libraries that are Not Multithread Safe

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

- User Interface
- DDE
- TCP
- Utility

- GPIB (if you are using a “Windows 3.1 compatibility” driver)
- Formatting and I/O (Standard I/O Window)
- ANSI C (Standard I/O Window)

Using LabWindows/CVI Libraries in External Compilers

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

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

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

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

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

You may also add the following DLL import library files from `cvi/extlib` to 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 */
```

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

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

If you are using 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 of the LabWindows/CVI libraries, except the ANSI C standard library. When you create an executable or DLL in an external compiler, you use the compiler's own ANSI C standard library. Because of this, you must use the external compiler's include files for the ANSI C library when compiling source files. Although the include files for the other LabWindows/CVI libraries are in the `cvi\include` directory, the LabWindows/CVI ANSI C include files are in the `cvi\include\ansi` directory. Thus, you can specify `cvi\include` as an include path in your external compiler while at the same time using the external compiler's version of the ANSI C include files.

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

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

Standard Input/Output Window

One effect of using the external compiler's ANSI C standard library, is that the `printf` and `scanf` functions do not use the LabWindows/CVI Standard Input/Output window. If you want to use `printf` and `scanf`, you must create a console application (called a character-mode executable 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 being loaded from the user interface resource (`.uir`) file. This is true whether you are running your program in the LabWindows/CVI development environment or as a standalone executable.

When you link your program in an external compiler, no such table is made available to the User Interface Library. To resolve callback references, you must use LabWindows/CVI to generate an object file containing the necessary table.

1. Create a LabWindows/CVI project containing the `.uir` files used by your program (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 to be generated. When you click on the **Create** button, the object file is generated with a table containing the names of all of the callback functions referenced in all of the `.uir` files in the currently loaded project. If the project is loaded and you modify and save any of these `.uir` files, the object file is regenerated to reflect the changes.

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. See the *`InitCVIRTE` and `CloseCVIRTE`* section later in this chapter.

Linking to Callback Functions Not Exported From a DLL

Normally, the User Interface Library searches for callback functions only in the table of functions in the executable. When you load a panel or menu bar from a DLL, you may want to link to non-static callback functions contained in, but not exported by, the DLL. You can do this by calling the `LoadPanelEx` and `LoadMenuBarEx` functions. When you pass the DLL module handle to `LoadPanelEx` and `LoadMenuBarEx`, the User Interface Library searches the table of callback functions contained in the DLL before searching the table contained in the executable. Refer to See 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, the table of functions is included in the DLL automatically. If you create your DLL using an external compiler, you must generate an object file containing the necessary table as follows.

1. Create a LabWindows/CVI project containing the `.uir` files loaded by your DLL (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 to be generated. When you click on the **Create** button, the object file is generated with a table containing the names of all of the callback functions referenced in all of the `.uir` files in the currently loaded project. If the project is loaded and you modify and save any of these `.uir` files, the object file is regenerated to reflect the changes.

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. See the *`InitCVIRTE` and `CloseCVIRTE`* section later in this chapter.

Resolving References from Modules Loaded at Run-Time

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

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

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

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

Resolving References to Non-LabWindows/CVI Symbols

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

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

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

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

Normally, the Utility Library `LoadExternalModule` function resolves run-time module references using only symbols in your executable or previously loaded run-time modules. When you load an object or static library file from a DLL, you may want to resolve references from that module using non-static symbols contained in, but not exported by, the DLL. You can do this by calling the `LoadExternalModuleEx` function. When you pass the DLL module handle to `LoadExternalModuleEx`, the Utility Library searches the symbol table contained in the DLL before searching the table contained in the executable. Refer to Chapter 8, *Utility Library*, of the *LabWindows/CVI Standard Libraries Reference Manual* for detailed information on `LoadExternalModuleEx`.

If you create your DLL in LabWindows/CVI, the table of symbols is included in the DLL automatically. If you create your DLL using an external compiler, no such table is made available to the Utility Library. Thus, when you are using an external compiler, you must include in your DLL one or more object files containing the necessary symbol tables. You can do this using the technique described in the preceding section, *Resolving References to Non-LabWindows/CVI Symbols*. You must call `InitCVIRTE` and `CloseCVIRTE` in your `DLLMain` function. See the *`InitCVIRTE` and `CloseCVIRTE`* section later in this chapter.

Run State Change Callbacks Are Not Available in External Compilers

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

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

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

Calling InitCVIRTE and CloseCVIRTE

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

For an executable using `main` as the entry point, your code should 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 should 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 need to call `CloseCVIRTE` in `DLLMain`. The code should 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 and NT.***

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

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

When you use an external compiler to create an object or library file for use in LabWindows/CVI, you must choose the compiler options carefully. For all compilers, LabWindows/CVI is designed to work with the default options as much as possible. In some cases, however, you need to choose options that override the defaults. Additionally, there may be some defaults which you must not override.

Microsoft Visual C/C++

LabWindows/CVI is compatible with all of the defaults.

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

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

Borland C/C++ command line compiler

LabWindows/CVI is compatible with all of the defaults.

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

`-a`	(Data Alignment)
`-K`	(Unsigned Characters)
`-u-`	(Turn Off Generation of Underscores)

`-N` (Test Stack Overflow)
`-p` (Pascal Calling Convention)
`-pr` (Register Calling Convention)
Correct Pentium FDIV Flaw

WATCOM C/C++

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

`-ei` (Force Enums to Type Int)
`-bt=nt` (target platform is Windows NT or Windows 95)
`-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 the function `LoadExternalModule` or `LoadExternalModuleEx`, you must also add the following compiler option:

`-d__NO_MATH_OPS`

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

`-Zp` (Structure Alignment)

Symantec C/C++

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

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

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

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

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

Creating Executables in LabWindows/CVI

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

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

Creating DLLs in LabWindows/CVI

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

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

There is no provision for debugging DLLs you compile in LabWindows/CVI. You can debug your project in LabWindows/CVI, before creating the DLL.

Customizing an Import Library

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

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

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

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

Preparing Source Code for Use in a DLL

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

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

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

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

Calling Convention for Exported Functions

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

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

The `__stdcall` keyword in not recognized on other platforms, such as UNIX or Windows 3.1. If you are working with source code that might be used on other platforms, you should use a

macro in place of __stdcall. The DLLSTDCALL macro is defined in the cvidef.h include file for this purpose.

The following are examples of using the DLLSTDCALL macro.

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

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

Exporting DLL Functions and Variables

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

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

Include File Method

You can 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 may contain nested #include statements, but the declarations in the nested include files are not exported. In the Create Dynamic Link Library dialog box, you select from a list of all of the include files in the project.

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

Export Qualifier Method

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

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

If the type of your variable or function requires an asterisk (*) in the syntax, put the qualifier after the asterisk, as in the following 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 have included in your DLL project an object or library file defining exported symbols, LabWindows/CVI cannot correctly create import libraries for each of the compilers it works with. This problem does not arise if you are using only source code files in your DLL project.***

Marking Imported Symbols in Include File Distributed with DLL

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

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

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

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

You can solve these problems in several different ways.

- You can avoid exporting variables from DLLs, and thereby eliminate the need to use import qualifiers. For each variable you want to export, you can create functions to get and set its value or a function to return a pointer to the variable. You do not need to use import qualifiers for functions. This is the simplest approach. (Unfortunately, it does not work if you use an export qualifier in a function definition and you are creating the DLL with an external compiler that requires the declaration and definition to agree.)
- You can create a separate include file for distribution with the DLL.

- You can use a special macro that resolves to either an import or export qualifier depending on a conditional compilation flag. In LabWindows/CVI you can set the flag in your DLL project by using the **Compiler Defines** command in the **Options** menu of the Project window.

Recommendations

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

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

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

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

Automatic Inclusion of Type Library Resource for Visual Basic

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

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

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

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

Creating Static Libraries in LabWindows/CVI

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

You need a separate project for each static library you want to create. You must select **Static Library** from the submenu attached to the **Target** command in the **Build** menu of the Project window. When **Static Library** is checkmarked, the **Create Static Library** command appears below the **Target** command in the **Build** menu. Refer to Chapter 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, all object modules from the*** `.lib` ***are included in the static library. When an executable or DLL is created, only the modules needed from the*** `.lib` ***file are used.***

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

Creating Object Files in LabWindows/CVI

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

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

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

Calling Windows SDK Functions in LabWindows/CVI

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

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 caused by function name and typedef conflicts between the Windows SDK and the LabWindows/CVI libraries. The LabWindows/CVI include files contain special macros and conditional compilation to adjust for declarations in the SDK include files. Thus, the SDK include files must be processed first, followed by the LabWindows/CVI include files.

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

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

Although there are a very large number of SDK include files, normally you need 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 which 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 is built on top of 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 there are no specific restrictions on using SDK functions in LabWindows/CVI, it is recommended 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 is not designed to handle multiple threads. For instance, if your main program terminates without destroying the threads, they are not terminated. Also, the LabWindows/CVI libraries are not multithread safe when called from a program linked in LabWindows/CVI.

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

Automatic Loading of SDK Import Libraries

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

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

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

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

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

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

Compiling in LabWindows/CVI for Linking in LabWindows/CVI

Use LabWindows/CVI's SDK and ANSI C include files. This happens automatically. You do not need to set up any special include paths; the include paths are set up automatically.

Compiling in LabWindows/CVI for Linking in an External Compiler

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

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

Compiling in an External Compiler for Linking in an External Compiler

Use the external compiler's SDK and ANSI C include file. 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 LabWindows/CVI's 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 and NT

In Windows 3.1, you can handle hardware interrupts in a DLL. In Windows 95, you must handle hardware interrupts in a VxD. In Windows NT, you must handle hardware interrupts in a kernel mode driver. You cannot create VxDs and kernel mode drivers in LabWindows/CVI. Instead, you must create them in Microsoft Visual C/C++, and you also need utilities available in the Microsoft Device Driver Developer Kit (DDK).

Under Windows 3.1, it is extremely difficult to call into LabWindows/CVI source code at interrupt time. Making such a call is easier under Windows 95 and NT. Under Windows 95 and NT, you can arrange for a function in your LabWindows/CVI source code to be called after your VxD (or kernel mode driver) interrupt service routine exits. You do this by creating a thread for

your interrupt callback function. The callback function executes a loop which blocks its thread until it is signaled by the interrupt service routine. Each time the interrupt service routine executes, it unblocks the callback thread. The callback thread then performs its processing and blocks again.

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

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

Chapter 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 generated by 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. The compiled module then can be used in any of the methods described in section *About Loadable Compiled Modules* 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, the LabWindows/CVI header files must be included 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 Developer 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 10.x IDE.

  – Set the Project Target environment to 32-bit Windows 3.x, and set the Image Type to Library[.lib]

  – Turn on **Disable stack depth checking [-s]**.

  – Turn on **Change char default to signed [-j]**.

  – Add the following to the **Other Options:** `-zw -d_NI_mswin16_`

  – Turn on **Generate as needed [-of]** for Stack Frames.

  – Turn on **No debugging information**.

  – Turn on **In-line with coprocessor [fpi87]** for Floating Point Model.

  – Turn on **Compiler default** for the Memory Model.

  – Turn on **80486 Stack based calling [-4s]** 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 may use optimization flags in addition to the `f`, and you may use other flags, such as `-wn,` which do not effect 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 structures with bit fields cannot be shared 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 may be called from within Borland compiled modules.

Note: ***This rule applies only to return values. You may use structures,*** `float`***s and*** `double`***s as output parameters without limitation.***

- ANSI C library functions `div` and `ldiv` return structures, and hence cannot be called from Borland compiled modules.
- The type `long double` is the same as `double` in CVI, while in Borland it is 10 bytes long, so objects of this type cannot be shared between Borland and CVI modules. This affects the `"%Le"`, `"%Lf"`, `"%Lg"` format specifiers of `printf`, `sprintf`, `fprintf`, `scanf`, `sscanf`, `fscanf`, and others.
- Because structures with bit fields cannot be shared between Borland and CVI, the macros in `stdio.h (getc, putc, fgetc, fputc)` cannot be used 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 will 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 **Allocate enums as ints**.
- Turn off **Fast floating point**.
- Use the C calling convention.

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

Use the following options when compiling 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 byte.
- Turn off Use Pascal Calling Convention.

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 may 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 need to include and possibly modify the glue code in a Watcom compiled module that must be loaded with the DLL.

The normal way of communicating with a DLL is by calling functions in the DLL. However, there are cases where you must use other communication methods. The most typical case is that of an interrupt service routine in a DLL that must notify the application that the interrupt occurred. This must be done through a callback function. 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. Both can be found in the **Run Options** menu of the Project window.

- Enable the **Check Disk Dates Before Each Run** option when you are iteratively modifying a DLL or DLL glue code file and running a LabWindows/CVI test program that calls into the DLL. By enabling the **Check Disk Dates Before Each Run** option, you ensure that the most recent version of the DLL and DLL glue code is linked 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. DLLs should be written so that they can remain in memory across multiple program executions. Nevertheless, if you are using 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. `Typedefs` for these types are also allowed.
- 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. `Typedefs` for these types are also allowed.
- 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 need to 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, you can 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 may experience a problem. The DLL may attempt to lock the buffer in memory by calling the Windows SDK function `GlobalPageLock`. `GlobalPageLock` fails on buffers allocated with the Watcom `malloc` function used by LabWindows/CVI 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 should 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 performing 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.dll */
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 are using Microsoft C to build the DLL, you may get a linker error for an undefined symbol*** `_acrtused2`. ***Include the following dummy function in your DLL to fix this error.***

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

This error occurs only in Microsoft C versions 7.00 and above. Also, when linking the DLL, specify `WIN87EM.LIB` ***as the first library to be linked.***

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 is automatically generated by LabWindows/CVI when the DLL is loaded. However, you *cannot* use this method in the following cases.

- The DLL requires special interface functions compiled outside of the DLL as a support module (`.obj` or `.lib`).
- You expect to pass arrays bigger than 64 K to functions in the DLL.
- The DLL uses a pointer passed to a function in the DLL after the function returns. For example, you pass an array to a function which starts an asynchronous I/O operation. The function returns immediately, but the DLL continues to operate on the array.
- You want to pass a function pointer to the DLL so that the DLL will call the function later. For example, the DLL makes a direct callback into 32-bit code.

- You want to 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, see the *DLLs That Cannot Use Glue Code Generated at Load Time* section of this chapter for details on how to proceed. Otherwise, see 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 associated with the DLL when it is loaded.

Any functions that are 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.)

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`.

Note: ***You may create an object module that contains the glue code. If you do so, LabWindows/CVI can load the DLL faster because it does not need 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, these must be combined with the glue code to form the `.obj` or `.lib` file.

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

The 32-bit Watcom compiled `.obj` or `.lib` file that is associated with the DLL is loaded first. For instance, if you want to include `x.dll` and `x.obj` in the project, 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 there is a `.dll` 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` for `.lib` file. If it cannot find the `.dll`, LabWindows/CVI looks for it as described in the documentation for the Windows SDK `LoadLibrary` function.

Alternatively, you can create a `.pth` file in the same directory as the `.obj` or `.lib` file with the same base name. LabWindows/CVI passes contents of the first line in the `.pth` file to the Windows SDK `LoadLibrary` function.

Rules for the DLL Include File Used to Generate Glue Code

You can generate the DLL glue source file by opening the `.h` file for the DLL in a Source window of LabWindows/CVI 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. This `.c` file must be modified as described in this section and compiled using the Watcom compiler. See the *Using 32-Bit Watcom Compiled Modules under Windows 3.1* section of this chapter for information on using the Watcom compiler with LabWindows/CVI.

If any of the functions in the DLL are declared as `PASCAL`, `pascal`, or `_pascal`, they must be declared as `PASCAL` in the `.h` file used 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 compiled under Watcom, calls to the function would be compiled incorrectly as if the function were `PASCAL`. You have two options.

- Have two separate `.h` files, one which includes the `PASCAL` keyword and one which 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 compiling.

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 of the DLL

Support modules contain special interface functions used by the DLL 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 there are link errors in the form of unresolved references, then 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 the DLL is Passed Arrays Bigger Than 64 K

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 would be 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 needed only if the DLL function performs floating point operations. They are innocuous and execute quickly, so 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`.

<u>If the DLL Retains a Buffer After the Function Returns (an Asynchronous Function)</u>

If the DLL retains a buffer after the function returns, you must modify the glue code source file. Suppose there is a function called `WriteRealArrayAsync` just like `WriteRealArray`, except that it returned before it completed the writing of the real array, and a function called `ClearAsyncWrite` was used to clean up 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. See the *Special Considerations when Using a Loadable Compiled Module* 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 __RunStateChangeCallback (int newState)
{
   if (newState == kRunState_Stop)
      ClearAsyncWrite ();
}
```

<u>If the DLL Calls Directly Back Into 32-Bit Code</u>

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, described in the *Recognizing Windows Messages Passed from a DLL* section of this chapter, it is not subject to the latencies of Window messaging.

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 Code** 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 generated by **Generate DLL Glue Code**.

The following example illustrates the glue code necessary if 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);
}
```

The following code shows the functions and data added or changed in the glue code generated from the header file for the DLL:

Note: ***Because a callback 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 needed 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 UserCallback. */
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;
   }
}
```

<u>If the DLL returns pointers</u>

DLLs return pointers that fall into the following two classes.

- Pointers pointing to memory that LabWindows/CVI allocates that are passed into the DLL and later returned by the DLL.

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

- Pointers pointing to memory that a DLL allocates.

  Because these pointers point to memory that is not in the LabWindows/CVI flat address space, the program cannot map them back into the normal 32-bit pointers used in LabWindows/CVI. 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 allocated in LabWindows/CVI. Notice that neither 16- nor 32-bit far pointers can be passed to LabWindows/CVI library functions and that LabWindows/CVI does not provide access to the far pointer manipulation functions provided by Watcom. You must write the loops to copy the data.

  Allocated data refers to data for which memory is allocated using the `malloc` function and to data that are static local variables.

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 need 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 need 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;
}
```

<u>If a DLL Is Passed a Pointer That Points to Other Pointers</u>

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 generated by LabWindows/CVI converts the pointer to the array `ptrs` to a 16-bit far pointer when it is passed 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 generated by LabWindows/CVI 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 Glue Code option 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 saved in the local variable in step 1.

For the functions `f` and `g`, the glue code generated by LabWindows/CVI 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 look like the following excerpt:

```
/* Assume NUM_ELEMENTS is the number of pointers in the array passed in.*/
/* 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 changed 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, you would 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, there are cases where other communication methods are needed. The most typical case is that of an interrupt service routine in a DLL that must notify the application that the interrupt occurred. You must do this 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. The three functions LabWindows/CVI uses to recognize Windows messages from a DLL are `RegisterWinMsgCallback`, `UnRegisterWinMsgCallback`, and `GetCVIWindowHandle`. You can call these functions from a Watcom compiled module or from source code.

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

RegisterWinMsgCallback

This function registers a callback function that LabWindows/CVI calls when it receives a Windows message. The declaration for this function follows.

```
unsigned short RegisterWinMsgCallback (WinMsgCallbackPtr callbackFunc,
                                       char *messageID, void *callbackData,
                                       long dataSize, int *callbackID,
                                       int deleteWhenUserProgramStops);
```

`typedef WinMsgCallbackPtr` is as follows:

```
typedef void (* WinMsgCallbackPtr) (unsigned short wParam,
                                    unsigned long lParam, void *callbackData);
```

The return value for `RegisterWinMsgCallback` is the message number that LabWindows/CVI expects. If the return value is zero, the registration failed. This number should be used in the DLL as the `uMsg` parameter for the Windows function `PostMessage`.

If **dataSize** is equal to zero, the **callbackData** pointer is the same pointer that will be passed to **callbackFunc** when it is called. If **dataSize** is greater than zero, the data pointed to by the **callbackData** pointer is copied and the **callbackFunc** will be called with the **callbackData** pointer pointing to the copy of the data.

Use the **messageId** string to generate the message number.

The **callbackId** receives the ID to be passed to `UnregisterWinMsgCallback`.

`deleteWhenUserProgramStops` determines whether the callback is automatically unregistered when the user program terminates. If the function returns a `1`, the callback is unregistered. If the function returns a `0`, the callback does not unregister.

wParam specifies 16 bits of additional message-dependent information. This number is the same as the **wParam** parameter for the Windows function `PostMessage`, in the DLL.

lParam specifies 32 bits of additional message-dependent information. This number is the same as the **lParam** parameter for the Windows function `PostMessage`, in the DLL.

UnRegisterWinMsgCallback

This function detaches a callback function that was previously registered through `RegisterWinMsgCallback`. The declaration for this function is as follows:

```
void UnRegisterWinMsgCallback (int callbackID);
```

callbackID is the ID stored in the reference parameter of `RegisterWinMsgCallback`.

GetCVIWindowHandle

This function returns the window handle associated with the LabWindows/CVI application. This number should be used in the DLL as the **hwnd** parameter for the Windows function, `PostMessage`. The declaration for this function is as follows:

```
int GetCVIWindowHandle(void);
```

To use this function, call `RegisterWinMsgCallback` and `GetCVIWindowHandle`. Pass their return values (`uMsg` and `hwnd`) 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 running 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. This can be remedied 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 that you want to call from outside the DLL must be `far`, exported, and must load the DS register. Loading the DS register is necessary if you are going to use any non-local variables in a function.
- Use the large or huge memory model. The savings gained by using smaller memory models is not worth having to use the `far` keyword throughout your code. This project option can be found in **Compiler » Memory Model » Segment Setup**.
- The DS segment can be loaded using the project option **SS!=DS, DS loaded on function entry**, which can be found in: **Compiler » Memory Model » Segment Setup.**
- If you try to use the optimize entry code option (`/GD`), which can be found in **Compiler » Windows » Prolog/Epilog » Generate Prolog/Epilog For**, it will conflict with the `/Au` option. You can either not use this option (set it to **None**), or insert `__loadds` in front of every function that you are exporting from the DLL.
- You can make the compiler export a function by either inserting `__export` between the return type and the function name, or 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 (pascal) or pre-append an underscore (cdecl).
- 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++

Be sure to consider the following issues or project options when you create a DLL with Borland C++ 4.*x*.

- Every function that you want to call from outside the DLL must be `far`, exported, and must load the DS register. Loading the DS register is necessary if you are going to use any non-local variables in a function.
- Use the large or huge memory model. The savings gained by using smaller memory models is not worth having to use the `far` keyword throughout your code. This project option can be found in **16-bit Compiler » Memory Model » Mixed Model Override.**

- You can make the compiler load the DS segment by either setting the project option **16-bit Compiler » Memory Model » Assume SS Equals DS** to **Never**, or by inserting `_loadds` in front of every function that you are exporting from the DLL.

- You can make the compiler export a function by either 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 (pascal) or pre-append an underscore (cdecl). 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** to **Byte** in the **16-bit Compiler » Processor** project options.

- Turn off **Case sensitive link** and **Case sensitive exports and imports** in the **Linker»General** project options.

- Do not use the **Linker goodies** options in the **Linker » 16-bit Linker** section of the project options.

DLL Search Precedence

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

- If it is associated with a `.fp` file, LabWindows/CVI finds the DLL using the following search precedence.

  1. If a `.pth` file with the same *full* path name as the `.fp` file is in the project, LabWindows/CVI loads the `.dll` file using the search method specified in the documentation of the Windows SDK documentation `LoadLibrary` function. The `.pth` file must contain the name of the `.dll` file, such as `mystuff.dll`. It should contain an absolute path or a simple file name.
  2. If a `.dll` file with the same *full* path name as the `.fp` file is in the project, LabWindows/CVI loads the `.dll` file using the absolute path of the `.dll` file in the project.
  3. If a `.pth` file with the same base name as the `.fp` file is in the same directory as the `.fp` file and there is not a `.lib` or `.obj` file of the same base name in the same directory, LabWindows/CVI loads the `.dll` file using the search method specified in the documentation of the Windows SDK `LoadLibrary` function. 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 is in the same directory as the `.fp` file, LabWindows/CVI loads the `.dll` file as long as it has the same base name as the `.fp` file and there is not a `.lib`, `.obj`, or `.pth` file of the same base name in the same directory.

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

   DLLs for VXI*plug&play* drivers are not in the same directory as the `.fp` files, but the directory containing the DLL is listed in the `PATH` environment variable. Therefore, step 5 makes it easier for you to use VXI*plug&play* instrument driver DLLs in LabWindows/CVI for Windows 3.1.

- If it is not associated with a file `.fp`, LabWindows/CVI finds the DLL using the following search precedence.

  1. If a `.pth` file is specified in the project list, then LabWindows/CVI finds the `.dll` using the search method specified in the documentation of the Windows SDK `LoadLibrary` function. The `.pth` file must contain the name of the `.dll` file, such as `mystuff.dll`. It should contain an absolute path or a simple file name.

  2. If the `.dll` file is specified in the project list, then LabWindows/CVI finds the `.dll` using the absolute pathname.

- If the `.dll` file is specified in a call to `LoadExternalModule`, then

  – If it is specified by an absolute pathname, LabWindows/CVI loads that file.

  – If it is specified by a relative pathname, then LabWindows/CVI searches for the `.dll` in the following order:

    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 specified in the **Include Paths** command in the **Options** menu of the Project window)

Chapter 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 generated by external compilers.

Calling Sun C Library Functions

You can call functions in the Sun C libraries from source code in LabWindows/CVI. LabWindows/CVI automatically links your program to the following libraries (in the `/usr/lib` directory) when they are needed:

Solaris 1: `libc.so`

Solaris 2: `libsocket.so`, `libnsl.so`, `libintl.so`, `libc.so`

In general, you can use the Sun header files (in the `/usr/include` directory) provided for these libraries. However, you should use the header files provided with LabWindows/CVI for the ANSI C functions.

Note: ***Compiler errors or warnings may result when you use some of the header files provided with Solaris 1 because they do not conform to the ANSI C Standard.***

Restrictions on Calling Sun C Library Functions

You cannot call any Sun C Library function that uses data types that are incompatible with the LabWindows/CVI compiler or libraries. In particular, you should 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 should not call any function that uses the `long long` data type. LabWindows/CVI does not recognize this non-ANSI type.

Creating Executables

Whether created by LabWindows/CVI or by an external compiler, executables that use the LabWindows/CVI libraries behave differently than programs that are run inside of LabWindows/CVI. This section describes these differences.

Run State Change Callbacks Are Not Available in Executables

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

You need the Run State Change Callback capability in LabWindows/CVI for the following reason: when you run a program in the LabWindows/CVI development environment, it is executed as part of the LabWindows/CVI process. When your program terminates, the operating system does not clean up as it does when a process terminates. LabWindows/CVI cleans up as much as it can, but your compiled module may need to do more. Also, if the program is suspended for debugging purposes, your compiled module may need to disable interrupts.

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

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 should always be 0 for UNIX applications. The second should be the same value as the second argument passed to your `main` function. `InitCVIRTE` returns 0 if it fails. You do not need to call this function when you are running your program in LabWindows/CVI because the libraries are already initialized. However, if you do not call this function, your executable will not work. For this reason, it is recommended 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` then your program may still work, but will have 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 may fail to find files.

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

Your use of externally compiled modules is restricted as follows.

- The objects must contain references only to symbols in the ANSI C Standard library or in the Sun C libraries that LabWindows/CVI loads (listed under Calling Sun C Library Functions).
- 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 differing sizes and values.
- You cannot load a Solaris 2 object file when running LabWindows/CVI under Solaris 1. However, you can load Solaris 1 objects when running under Solaris 2.

Compiling Modules With External Compilers

You can compile external modules using LabWindows/CVI header files instead of the headers supplied with the compiler. 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 flag `-nostdinc` to disable the standard include files and the flag `-I`*LabWindows/CVIHeadersPath* (where *LabWindows/CVIHeadersPath* is the path for your include files) to enable the LabWindows/CVI include files. Also, you should use the `-ansi` flag to accept ANSI C. 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
```

Some warnings about conflicting types of built-in functions `memcmp` and `memcpy` may be generated, 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 installed 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 the entire 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: ***The effective user ID of the LabWindows/CVI process must be superuser to use the*** `plock` ***function.***

Chapter 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, Windows 95 and Windows 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. In general, the portability of a LabWindows/CVI application can be assured by following a few simple guidelines.

- Write code in strict ANSI C.
- Observe and repair all LabWindows/CVI compile, link, and runtime diagnostics.
- Use LabWindows/CVI supported library functions, avoiding system dependent library calls when possible.
- Avoid the use of 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.

Any platform dependent code should be segregated in your source code using conditional preprocessor directives controlled with 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

Although LabWindows/CVI for Windows 95 and NT allows use of the Windows 32-bit SDK library calls, their use is discouraged unless you intend the LabWindows/CVI application to run only under Windows 95 and NT.

The `sopen` and `fdopen` functions are only available under Windows. Their use is discouraged unless you intend the LabWindows/CVI application to run only under Windows.

Although LabWindows/CVI allows use of host system library calls (such as `ioctl`, `fcntl`, and so on) under UNIX, their use is discouraged unless you intend the LabWindows/CVI application to run only under UNIX. In general, you should avoid using UNIX host system calls in C program files to ensure that your program is portable. See the *Calling the UNIX C Library from Source Code* section of Chapter 1, *LabWindows/CVI Compiler*, for more information on using user 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. See the *Calling the UNIX C Library from Source Code* section of Chapter 1, *LabWindows/CVI Compiler*, for more information on using UNIX C low-level functions. These functions are portable to Windows if you include `lowlvlio.h` in your Windows application.

The ANSI C, User Interface, Analysis, Formatting and I/O, Utility, GPIB, VXI, RS-232, and TCP libraries are portable across platforms.

Only LabWindows/CVI for Windows has DDE and Data Acquisition libraries. The X Property Library is only available under UNIX.

Although LabWindows/CVI allows TCP library calls from the application 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 are working with 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", prog_buf);
```

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

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

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

Externally Compiled Module Issues

Although you can use externally compiled modules in LabWindows/CVI as described in this manual, 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. Additionally, although SPARCstations have the same computer architecture, Solaris 1.*x* (Sun OS 4.*x*) and Solaris 2.*x* use different object file formats, which 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

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

Image file formats other than PCX (`.pcx`) may not be 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 there may exist some relative size differences. 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. The width of the National Instruments Meta Fonts may differ slightly from one platform to another, however. Allow for extra space in the width of all control labels to assure consistent appearance.

You may 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 may differ between the Windows and UNIX platforms. This happens because of differences between the underlying window management systems on which LabWindows/CVI is built.

You should not assign the <FORWARD DELETE> key as a hot-key in your user interface, because that key does not exist on all UNIX workstations.

Chapter 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 needed to run executables or use DLLs created 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, the LabWindows/CVI Run-time Engine can be bundled 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 and NT

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

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

No Math Coprocessor Required for Windows 95 and NT

You do not need a math coprocessor or emulator to use the LabWindows/CVI run-time libraries in Windows 95 or NT.

Minimum System Requirements for Windows 3.1

To run a standalone executable created by 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 using at least a 25 MHz 80386 or higher microprocessor (National Instruments recommends a 33 MHz 80486 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 accomodate your executable and any files the executable needs

Math Coprocessor Software Emulation for Windows 3.1

To run a standalone executable created by 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 file names in calls to these functions. If you use a relative path, the path is relative to the directory containing the executable. See the section *Location of Files on the Target Machine for Running Executable Programs and DLLs* 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 containing your executable program and associated files execute the shell script `makepkg` 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 9 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, but it 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 the 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 need 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 be logged in as root. There are two methods for installing and removing a package.

   Method 1: Use the Software Management Tool `swntool` located in the `/usr/sbin` directory of your system.

   Method 2: 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 or 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 file names in calls to these functions. If you use a relative path, the path is relative to the directory containing the executable. See the section entitled *Location of Files on the Target Machine for Running Executable Programs and DLLs* 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. Test your program by running 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, which is required to run your executable.

4. Make a copy of the installation script `INSTALL.sample` in the `misc/bin` directory and customize it using the information provided by `makedist`. 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 System. 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 need to provide users with the package file created by `makedist` and the customized installation script that will extract the files from the package.

Minimum System Requirements for UNIX

To run a standalone executable created by LabWindows/CVI for UNIX, your system must have the following:

- Sun SPARCstation
- Solaris 1.x (SunOS 4.1.2 or greater) or Solaris 2.4 (Sun OS 5.4)
- At least 24 megabytes of RAM
- At least 32 megabytes of disk swap space
- Free hard disk space equal to 2 MB, plus space to accomodate your executable and any files the executable needs

Configuring the Run-Time Engine

This section applies to you, the developer, as well as the end-user who uses your executable program. Feel free to use the text in this section in the documentation for your executable program.

Translating the Message File

The message file (`msgrt`*n*`.txt` where *n* is the version number of the Run-time Engine) is a text file containing the error messages displayed by the Run-time Engine. It is found 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 contained in quotation marks. You must not add or delete any message numbers.

3. Input the file into the `countmsg.exe` (`countmsg` on UNIX) utility so that it is encoded for use with the Run-time Engine, as in the following example:

   ```
   countmsg msgrt4.txt
   ```

Option Descriptions

The Run-time Engine recognizes the following options. On the PC, these options are set during the installation 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 the executable loads and executes the Run-time Engine, it must be able to locate the Run-time Engine on the hard disk. Under Windows 3.1, the executable finds the Run-time Engine using the `cvirtx` (where *x* is the version number of the Run-time Engine) configuration option. Set this configuration option as follows.

Note: ***Under Windows 95 and NT, the location of the Run-time Engine DLL is always the Windows system directory.***

- Windows 3.1 Configuration

  Set Windows 3.1 configuration options in the `win.ini` file. There is a configuration string associated with the `cvirtx` option in the `[cvirtx]` section as in the following example.

  ```
  [cvirt4]
  cvirt4=c:\cvi\cvirt4.exe
  ```

cvidir (Windows Only)

Under Windows 95 and NT, `cvidir` specifies the location of the directory containing the `bin` and `fonts` subdirectories that are required by the Run-time Engine. You must define this Registry option to enable the Windows 95 and NT Run-time Engine DLL to load.

For Windows 3.1, set the `cvidir` only if the Run-time Engine resides in a directory other than the directory containing the `bin` and `fonts` subdirectories. If you do not specify a directory, the Run-time Engine's default directory is the directory specified by `cvirtx` (where *x* is the version number of the Run-time Engine).

- Windows 95 and NT Configuration

  Set configuration options in Windows 95 and NT in the Registry under the following key.

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

  Change the path in the subkey, for example, from `\4.0\cvidir` to `c:\cvi`.

- Windows 3.1 Configuration

Set Windows 3.1 configuration options in the `win.ini` file. There is a configuration string associated with the `cvidir` option in the `[cvirtx]` section, similar to the following example.

```
[cvirt4]
cvidir=c:\cvi
```

Necessary Files for Running Executable Programs

In order for your executable to run successfully on a target computer, any files that are required by the executable must be accessible. Your final distribution kit should contain all of the necessary files for installing your LabWindows/CVI executable program on a target machine as shown in Figure 7-1.

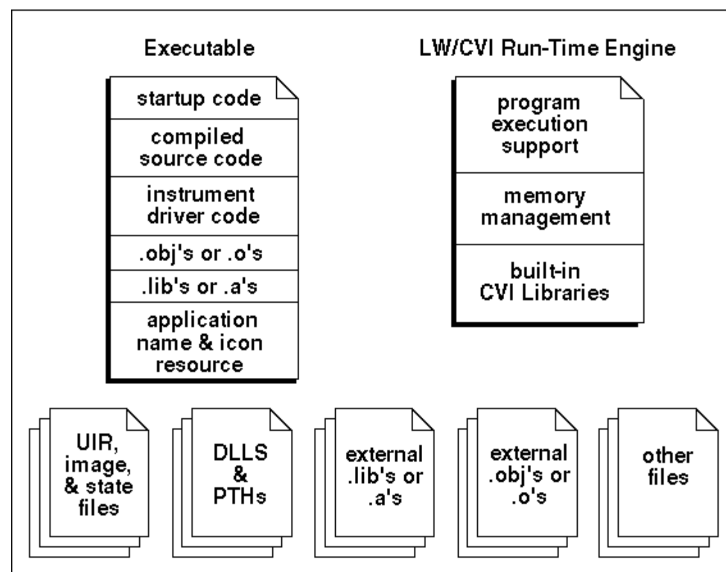


Figure 7-1. Files Needed to Run a LabWindows/CVI Executable Program on a Target Machine

- The **Executable** contains a precompiled, prelinked version of your LabWindows/CVI project and any instrument driver program files that are linked to your project. It also contains the application name and icon resource to be registered to the operating system. The executable has an associated icon that you can double-click on to start the application. When the executable is started, it loads and starts the Run-time Engine. Under UNIX, the executable returns the value returned by `main` or the value passed to `exit`.

- The **Run-Time Engine** (`CVIRT.DLL` and `CVIRTE.DLL` under Windows 95 and NT, `CVIRTn.EXE` under Windows 3.1, and `cvirtn` under UNIX, where *n* is the version of the Run-time Engine) is an execute-only version of the LabWindows/CVI environment. The Run-time Engine contains all of the built-in library, memory, and program execution help present in the LabWindows/CVI environment, without all of the program development tools such as the source editor, compiler, debugger, and user interface editor. The Run-time Engine is smaller than LabWindows/CVI environment and thus loads faster and requires less

memory. You need only one copy of the Run-time Engine on each target machine even when you have multiple executables.

- **UIR Files** are the User Interface Resource files that are used by your application program. Load these files using `LoadPanel` and `LoadMenuBar`.
- **Image Files** are the graphical image files that are programmatically loaded and displayed on your user interface using `DisplayImageFile`.
- **State Files** are the user interface panel state files that are saved using `SavePanelState` and loaded using `RecallPanelState`.
- **DLL Files**(Windows Only) are the Windows Dynamic Link Library files that are used by your application program.
- **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** are compiled 32-bit `.lib` files on the PC or `.a` files under UNIX that are loaded by `LoadExternalModule` and are not listed in the project.
- **External .obj or .o Files** are compiled 32-bit `.obj` files on the PC or `.o` files under UNIX that are loaded by `LoadExternalModule` and are not listed in the project.
- **Other Files** are files opened by your executable using `open`, `fopen`, `OpenFile`, and so on.

Necessary Files for Using DLLs Created in Windows 95/NT

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

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 be in 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. (See 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 on Windows 95/NT

For Windows 95 and NT, the following set of DLLs contain the run-time libraries.

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

These DLLs are distributed on a separate diskette (or in a separate directory in the CD-ROM) and are installed as part of LabWindows/CVI. The **Create Distribution Kit** command in the **Build** menu of the Project window optionally bundles the run-time library DLLs into your distribution kit. Alternatively, you can make copies of this diskette (or the CD-ROM directory) for separate distribution. The run-time library DLLs are always installed in the Windows system directory.

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

LabWindows/CVI Run-Time Engine on Windows 3.1

For Windows 3.1, the LabWindows/CVI Run-time Engine comes in the form of an executable file. The Run-time Engine is distributed with LabWindows/CVI on a separate diskette and is installed as part of the LabWindows/CVI installation. The **Create Distribution Kit** command in the **Build** menu of the Project window optionally bundles the Run-time Engine into your distribution kit. Alternatively, you can make copies of this diskette for separate distribution. The end-user selects the directory into which the Run-time Engine is installed.

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

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 (that is, 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 containing the executable.

Rules for Using DLL Files under Windows 95 and NT

In Windows 95 or NT, your executable or DLL can link to a DLL only via an import library. (In this section, a DLL used by an executable or by another DLL is referred to as a *subsidiary* DLL.) An import library can be linked into your program in any of the following ways.

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

If a DLL import library is listed in the project or is associated with an instrument driver or user library, the import library is statically linked into your executable or DLL. On the other hand, if the import library is loaded via a call to `LoadExternalModule`, it must be distributed separately from your executable and loaded at run time. See the section *Rules for Loading Files Using LoadExternalModule*.

Regardless of the method you use to link the import library, the subsidiary DLL must be distributed separately and is loaded at run time. 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 is described in the Windows SDK documentation for the `LoadLibrary` function. The search precedence is,

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

Create Distribution Kit automatically includes DLLs in your distribution kit that are referenced by true import libraries in your project. You must add to the distribution kit any DLLs that are loaded via `LoadExternalModule` or that you load by calling the Windows SDK `LoadLibrary` function.

DLLs for National Instruments hardware should not be part of your distribution kit. These DLLs must be installed from the distribution disks supplied by National Instruments.

Rules for Using DLL Files in Windows 3.1

DLL files and DLL path files are never linked into the executable, so you must distribute them as separate files. **Create Distribution Kit** automatically includes DLLs that are referenced by your project in your distribution kit. The only exceptions are DLLs for National Instruments hardware and DLLs that are loaded using `LoadExternalModule`.

DLLs for National Instruments hardware should not be part of your distribution kit. These DLLs must be installed from the distribution disks supplied by National Instruments.

If you are loading DLLs using `LoadExternalModule`, refer to the following section, *Rules for Loading Files Using LoadExternalModule*.

If you are using a DLL file, a DLL path file, or a DLL glue object module in your project or as a loaded instrument driver, the Run-time Engine always looks for a corresponding DLL path (`.pth`) file before looking for the DLL itself. This search mechanism lets the end-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 that has 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 search method specified in the documentation for the Windows SDK `LoadLibrary` function to find the DLL (`\WINDOWS`, `\WINDOWS\SYSTEM`, then the `PATH` environment variable).

2. Look for a `.dll` file in the directory of the executable that has the same base name as the file in the project or as the instrument driver. If the `.dll` file is not in that directory, use the search method specified in the documentation for the Windows SDK `LoadLibrary` function to find the DLL (`\WINDOWS`, `\WINDOWS\SYSTEM`, then the `PATH` environment variable).

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 developing 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

The following file types can be loaded by `LoadExternalModule`.

Library Files:	`.lib` (Windows) or `.a` (UNIX)
Object Modules:	`.obj` (Windows) or `.o` (UNIX)
DLL Import Library Files:	`.lib` (Windows 95 and 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 Referenced by External Modules 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 they are referenced elsewhere in your project. However, when you create a standalone executable, only modules that your project references are included in the executable. If an external module references modules not included in the executable, calls to `RunExternalModule` or `GetExternalModuleAddr` on that external module will fail.

To avoid this problem, you must force any missing modules into your executable or DLL. You can do this when creating your executable or DLL by using the **Add Files To Executable** or **Add Files To DLL** button to bring up a list of project `.lib`, project `.a`, Instrument, and Library files. Checkmark the files you want to be included in your executable or DLL. If a `.lib` or `.a` file is checkmarked, it will be 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. However, when you create an executable or DLL from your project, you may 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), see *Rules for Using DLL Files in Windows 3.1*.
  - For `.lib` files, by default, **Create Standalone Executable File** or **Create Dynamic Link Library** only links in the library modules that are referenced statically in the project. Therefore, *you* must force the modules containing the functions that are called through `GetExternalModuleAddr` to be linked into the executable.

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

    - Specify that the entire library file be linked into the executable by selecting it using 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.

    - Cause the modules you need to be linked into the executable by referencing them statically. For example, you could have an array of void pointers and initialize them to the names of the symbols needed.

  If you choose to link the library file directly into the executable, you must pass the simple filename of the library file to `LoadExternalModule`.

- If you *link in an external compiler* on 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, calls to `GetExternalModuleAddr` or `RunExternalModule` on modules that are linked into your executable or DLL will fail. Your alternatives are,

  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 containing a table of symbols you want to be found by `GetExternalModuleAddr`. If you use this method, pass the empty string (`""`) to `LoadExternalModule` as the module pathname. The empty string indicates that the module was linked into your project executable or DLL.

Using LoadExternalModule on Library and Object Files Not in the Project

When you call `LoadExternalModule` on a library or object file not in the project, you must keep the library or object file as a separately distributed file.

When you keep an object or library file separate, you can manage memory more efficiently and replace it without having to replace the executable or DLL. For this reason, it is recommended that if you are calling `LoadExternalModule` on a library or object in the project, you remove (or exclude) the file from the project before selecting **Create Standalone Executable File** or **Create Dynamic Link Library**, and then include it as a separate file when using **Create Distribution Kit**.

However, bear in mind 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 recommended that you place the file in the same directory as the executable or DLL. By placing the file in the same directory, you are able to 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 containing the executable or DLL.

Using LoadExternalModule on DLL Files under Windows 95 and NT

Under Windows 95 and NT, DLLs cannot be directly referenced by `LoadExternalModule`. The call to `LoadExternalModule` must reference the DLL import library instead. The import library can be linked into your executable or DLL, or it can be distributed separately and loaded dynamically. For import libraries that are linked into your executable or DLL, see *Using LoadExternalModule on Files in the Project*. For import libraries that are loaded dynamically, see the section, *Using LoadExternalModule on Library and Object Files Not in the Project.*

DLLs must always be distributed as separate files. The operating system finds the DLL associated with the loaded import library using the standard DLL search algorithm, which is described in the Windows SDK documentation for the `LoadLibrary` function. The search precedence is:

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

Using LoadExternalModule on DLL and Path Files under Windows 3.1

DLL files and DLL path files are never linked into the executable, so they must be distributed as separate files.

Your executable can call `LoadExternalModule` directly on a DLL or DLL path file only if the DLL or DLL path file is included in the project. When you select **Create Standalone Executable File**, the DLL glue code is created automatically and linked into the executable.

Alternatively, you can pass the DLL glue object module file name for the DLL 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, `LoadExternalModule` must be passed 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 that has the same base name as the file 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 search method specified in the documentation for the Windows SDK `LoadLibrary`

function to find the DLL (`\WINDOWS`, `\WINDOWS\SYSTEM`, then the `PATH` environment variable).

2. Look for a `.dll` file in the directory of the executable that has the same base name as the file passed to `LoadExternalModule`. If the `.dll` file is not in that directory, use the search method specified in the documentation for the Windows SDK `LoadLibrary` function to find the DLL (`\WINDOWS`, `\WINDOWS\SYSTEM`, then the `PATH` environment variable).

If you maintain the DLL glue object module as a separate file from the executable, `LoadExternalModule` must be passed 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 search method specified in the documentation for the Windows SDK `LoadLibrary` function to find the DLL (`\WINDOWS`, `\WINDOWS\SYSTEM`, then the `PATH` environment variable).

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. If the `.dll` file is not in that directory, use the search method specified in the documentation for the Windows SDK `LoadLibrary` function to find the DLL (`\WINDOWS`, `\WINDOWS\SYSTEM`, then the `PATH` environment variable).

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 developing 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. The source file is automatically compiled and linked 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 are using an external compiler, see the section, *Using LoadExternalModule on Files in the Project*.

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.
- Make sure the file is linked into the project by directly referencing it from the project.

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

Rules for Accessing Other Files

The functions for accessing files (`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 normally is the directory of the executable. However, if there is a different directory in the Working Directory field of the Properties dialog box for the executable, then that is the initial current working directory. Under UNIX, the initial current working directory is the directory from which you invoked 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

You typically enable debugging and **Break on Library Errors** while you develop your application in LabWindows/CVI. With these utilities enabled, LabWindows/CVI checks for programming errors in your source code, so you may have a tendency to be relaxed in your own error checking.

However, when you create a standalone executable program or DLL, all of your source modules are compiled. Compiled modules have debugging and **Break on Library Errors** disabled, resulting in smaller and faster code. Thus, you must perform your own error checking when you are creating a standalone executable program or DLL. Refer to Appendix B, *Error Checking in LabWindows/CVI*, for details about performing error checking in your code.

Chapter 8
Distributing Libraries and Function Panels

This chapter describes how to distribute libraries, how to add libraries to an end-user's **Library** menu, and how to 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. For any library program file that is purely a support file for the other files—in other words, it contains no user-callable function—you can prevent it from appearing in the **Library** menu by creating a `.fp` file with no classes or function panel windows.

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 on Windows 3.1, `.cvi.ini` on UNIX, or the Registry on Windows 95 and NT.

Under Windows 3.1 and UNIX, the `modini` program is included in the LabWindows/CVI `bin` subdirectory for this purpose. A documentation file (`modini.doc`) is also included, as is the source code.

Under Windows 95 and NT, the `modreg` program is included in the LabWindows/CVI `bin` subdirectory for this purpose. A documentation file (`modreg.doc`) is also included, as is the source code.

Assume that you install function panels for 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 should be,

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

After the library files are installed, the `modini` program should 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 and NT, your `modreg` command file should 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 library files are installed, the `modreg` program should 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, your changes will be lost.***

Specifying Library Dependencies

When one library you are distributing library is dependent upon the other libraries in your set, you can specify this dependency in the function panel file for the dependent library. When loading the dependent library, LabWindows/CVI 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 *Instrument Driver Developers Guide, Chapter 2, Function Tree Editor*, for details on this command).

LabWindows/CVI can find the required libraries most easily when all of them are 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. Alternatively, you can add to the Instrument Directories list by editing `cvi.ini` file under Windows 3.1, `.cvi.ini` under UNIX, or the Registry under Windows 95 and NT.

The recommended approach is for your installation program to add to `cvi.ini` (or `.cvi.ini` or the Registry) automatically. Under Windows 3.1, the `modini` program is included in the LabWindows/CVI `bin` subdirectory for this purpose. A documentation file (`modini.doc`) is also included, as is the source code.

Under Windows 95 and NT, the `modreg` program is included in the LabWindows/CVI `bin` subdirectory for this purpose. A documentation file (`modreg.doc`) is also included, as is the source code.

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

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

If `liba` depends on `libb`, you need to 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 should be,

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

After the library files are installed, the `modini` program should 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 and NT, your `modreg` command file should be,

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

After the library files are installed, the `modreg` program should 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 runing while you use these programs, your changes will be lost.***

Appendix 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 Runtime Error	Ensure that the correct format specifier is used, and that there are not extra characters 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	The 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	The `#line` preprocessor directive requires a non-zero (_0) line number value specified.
`#line directive cannot specify line greater than 32767.`	Compile Error	The `#line` preprocessor directive cannot set the line greater than 32767.
`#line directive expects numeric argument.`	Compile Error	The `#line` preprocessor directive requires a line number value to be specified following `#line`.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`## at beginning of macro definition.`	Compile Error	The `##` preprocessing token was found at the beginning of a macro definition. Ensure that `##` is preceded by a preprocessing token.
`## at end of macro definition.`	Compile Error	The `##` preprocessing token was found at the end of a macro definition. Ensure that `##` is followed by a preprocessing token(s).
`, or ) expected.`	Compile Error	Ensure that the function macro argument list is terminated with a `)` or that all of the macro arguments are separated by `,`.
`0 flag is not valid with c, s, p, and n modifiers.`	Non-Fatal Runtime Error	This error may be caused by the use of an incorrect format specifier, or by the use of a field width starting with `0`.
NAME `is a predefined macro and cannot be the subject of an #undef.`	Compile Error	Make sure that the name specified for the `#undef` preprocessor directive is not that of a predefined macro.
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 specified for the bit field >= 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 (tabs count as 1).Use the editor in which the file was created to split the line.
TYPE `is an illegal bit field type.`	Compile Error	Only `int` and `unsigned` types are allowed for bit field declarations; ensure that one of these types is being used.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`TYPE used as an lvalue.`	Compile Warning	A type that cannot be modified is being used as the target of an assignment. Probably caused by an lvalue that is a dereference of an object declared as `(void *)`.
`Aborted load of library FILE.`	Link Error	The library load operation was aborted. A more specific diagnostic of the library load error should have preceded this message.
`Aborted load of member NAME from library FILE.`	Link Error	The library load of a member was aborted. A more specific diagnostic of the library load member error should have preceded this message.
`Aborted load of object module FILE.`	Link Error	The object file load was aborted. A more specific diagnostic of the object file load error should have preceded this message.
`Absolute segments not supported: segment name NAME.`	PC/Windows Load Error	OMF object file contains a segment to be loaded at an absolute address.
`Anonymous enum declared inside parameter list has scope only for this declaration.`	Compile Warning	The 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 should 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	The 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 should declare the structure type before declaring function types that use it.
`Anonymous union declared inside parameter list has scope only for this declaration.`	Compile Warning	The 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 declaring function types that use it.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Argument 4 must be 0 or 1.`	Fatal Runtime Error	The value of the argument to the library function must be `0` or `1`.
`Argument` *`NUMBER`* `must be 0, 1 or 2.`	Fatal Runtime Error	The 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 Runtime Error	The argument to the function is not a pointer to the expected type of callback function.
`Argument must be an open stream.`	Fatal Runtime Error	The argument to the I/O library function must be one of the standard streams (`stdin`, `stdout`, `stderr`) or a stream opened with the functions `fopen()` or `freopen()`.
`Argument must be character.`	Fatal Runtime Error	The value of the argument to the library function must be less than 256.
`Array argument too small.`	Fatal Runtime Error	The library function called requires an array that is larger than the specified argument. Check that the array was either declared or allocated with sufficient elements for the function call.
`Array argument too small (`*`NUMBER`*` bytes). Argument must contain at least `*`NUMBER`*` bytes (`*`NUMBER`*` elements).`	Fatal Runtime Error	The library function called requires an array that is larger than the specified argument. Check that the array was either declared or allocated with the number of elements reported by this error message.
`Array index (`*`NUMBER`*`) too large (maximum: `*`NUMBER`*`).`	Non-Fatal Runtime Error	The array was indexed past the last element.
`Assertion error: `*`EXPRESSION`*`.`	Fatal Runtime Error	The value of the argument *`EXPRESSION`* to the Standard C Library macro `assert` is 0.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
Assignment between *TYPE* and *TYPE* is compiler-dependent.	Compile Warning	Although allowed, caution is advised because an assignment of an integer type expression value to an enum type target may not correspond to any known enumeration constant for that enum type. Depending on the enumeration, the size of the enum type may be 1, 2 or 4 bytes and therefore may be incapable of representing all integer values.
Assignment of invalid pointer value.	Non-Fatal Runtime Error	The value being assigned to a pointer is an invalid pointer value. Check the right hand side of the assignment to determine if is the result of a previous invalid pointer operation.
Assignment of out-of-bounds pointer: *NUMBER* bytes before start of array.	Non-Fatal Runtime Error	The value being assigned to the pointer refers to an invalid location, which is *NUMBER* bytes before an array. The right hand 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 Runtime Error	The value being assigned to the pointer refers to an invalid location, which is *NUMBER* bytes past the end of an array. The right hand side of the assignment is probably the result of previous illegal pointer arithmetic.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Assignment of pointer to freed memory.`	Non-Fatal Runtime Error	The value being assigned to the pointer is invalid because it refers to a location in dynamic memory that was deallocated with the function `free`. Once memory is freed, all pointers into that block of memory become invalid.
`Assignment of uninitialized pointer value.`	Non-Fatal Runtime Error	The value being assigned to the pointer is invalid because it was not initialized. The right hand side of the assignment is probably an uninitialized local variable or an object in dynamic memory that was allocated with `malloc`. Initialize local variables and dynamic memory before you use them. The function `calloc` both allocates and initializes dynamic memory.
`Assignment to const identifier` *`NAME`*`.`	Compile Error	`const` declared variables or parameters are treated as read only values that cannot be modified once initialized; Ensure that the identifier is not being modified by an assignment operation.
`Assignment to const location.`	Compile Error	`const` declared variables or parameters are treated as read only values that cannot be modified once initialized; Ensure that the `lvalue` (such as an array reference, or a pointer dereference) specifying the `const` location is not being modified by an assignment operation.
`Attempt to free invalid pointer expression.`	Fatal Runtime Error	The pointer value passed to the function `free` is invalid. It is probably the result of a previous invalid pointer operation.
`Attempt to free pointer to freed memory.`	Fatal Runtime Error	The pointer value passed to the function `free` refers to a location in dynamic memory that was already deallocated.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Attempt to free uninitialized pointer.`	Fatal Runtime Error	The pointer value passed to the function `free` is invalid because it was not initialized. It is probably an uninitialized local variable. Initialize local variables before you use them.
`Attempt to read beyond end of array.`	Non-Fatal Runtime Error	The source array was not large enough to satisfy the destination specifiers.
`Attempt to read beyond end of string.`	Non-Fatal Runtime Error	The source string is not large enough to satisfy the destination specifiers.
`Attempt to realloc invalid pointer expression.`	Fatal Runtime Error	The pointer value passed to the function `realloc` is invalid. It is probably the result of a previous invalid pointer operation.
`Attempt to realloc pointer to freed memory.`	Fatal Runtime Error	The pointer value passed to the function `realloc` refers to a location in dynamic memory that was already deallocated.
`Attempt to realloc uninitialized pointer.`	Fatal Runtime Error	The pointer value passed to the function `realloc` is invalid because it was not initialized. It is probably an uninitialized local variable. Local variables must be initialized before they are used.
`Attempt to write beyond end of array.`	Non-Fatal Runtime Error	The output array is smaller than required by the given format specifiers and input parameters.
`Attempt to write beyond end of string.`	Non-Fatal Runtime Error	The output string is smaller than required by the given format specifiers and input parameters.
`b modifier must precede o modifier.`	Non-Fatal Runtime Error	If both the `b` and `o` modifiers are present, then the `b` modifier must precede the `o` modifier.
`Bad BSS section encountered while reading external module: FILE.`	Object Load Error	The object module is corrupted or is of a type that cannot be loaded into LabWindows/CVI.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Bad COFF Library header.`	Object Load Error	The library file you are loading is either corrupted or not in the COFF format.
`Bad COFF Library member header.`	Object Load Error	The COFF library you are loading contains a module that is corrupted or in an invalid format.
`Bad location code: OMF record position` *NUMBER*`: OMF record type` *NAME*.	Link Error	The object module has been corrupted or is of a type that cannot be loaded into CVI.
`Bad magic number encountered while reading external module:` *FILE*.	Link Error	The object module has been corrupted or is of a type that cannot be loaded into CVI.
`Bad method: OMF record position` *NUMBER*`: OMF record type` *NAME*.	Link Error	The object module has been corrupted or is of a type that cannot be loaded into CVI.
`Bad name: OMF record position` *NUMBER*`: OMF record type` *NAME*.	Link Error	The object module has been corrupted or is of a type that cannot be loaded into 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 supported by LabWindows/CVI.
`Bad relocation record encountered while reading external module:` *FILE*.	Link Error	The object module has been corrupted or is of a type that cannot be loaded into 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 supported by LabWindows/CVI.
`Byte ordering is invalid.`	Non-Fatal Runtime Error	The byte ordering specified by the `o` modifier 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 -1 to size of the integer -1.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`c modifier valid only with l format specifier.`	Non-Fatal Runtime Error	The `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 Runtime Error	The function *NAME* was specified as a callback function for an item in a user interface resource file, but 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 Runtime Error	The function *NAME* was specified as a callback function for an item in a user interface resource file, but the function does not exist.
`Calling conventions have no effect on variables; calling convention ignored. The position of the calling convention modifier may be incorrect.`	Compile Warning	A calling convention keyword was placed before a variable name. For function pointers, the calling convention must be placed to the left of the "*", e.g. `int (__cdecl * funptr)();`
`Cannot concatenate wide and regular string literals.`	Compile Warning	Make sure the string literals being concatenated are either both wide string literals or regular string literals.
`Cannot free: memory not allocated by malloc() or calloc().`	Fatal Runtime Error	The pointer value passed to the function `free` is invalid because it does not point to dynamic memory allocated by `malloc` or `calloc`. Only pointers obtained by a call to one of these two functions can be deallocated using `free`.
`Cannot generate glue for a function without a prototype:` *`NAME`*`.`	Glue Code Generation Error	In order to generate glue code for a DLL function, the function must be specified by a complete prototype. The types of the parameters must be specified in the prototype.
`Cannot generate glue for a static function:` *`FUNCTION`*`.`	Glue Code Generation Error	Static functions in a DLL cannot be exported; so it is useless to generate glue code for them.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Cannot generate glue for a variable argument function:` *`FUNCTION`*`.`	Glue Code Generation Error	You cannot use DLL functions that accept a variable number of arguments with CVI.
`Cannot initialize undefined` *`TYPE`*`.`	Compile Error	An attempt was made to initialize a declaration of an incomplete `struct` or `union` type, such as a `struct` or `union` type whose members have not yet been 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	A variable that you have declared as `extern` is defined in a DLL import library, but you did include the `__import` qualifier in the declaration.
`Cannot link variable` *`NAME`* `to import library without declspec(dllimport) keyword in declaration.`	Link Error	A variable that you have declared as `extern` is defined in a DLL import library, but you did include the `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 of a constant integer expression.
`Cast from` *`TYPE`* `to` *`TYPE`* `is illegal in constant expressions.`	Compile Error	Constant expression values must be determinable at compile time; make sure that the cast operation does not involve a pointer type as storage has not yet been allocated at compile time.
`Cast from` *`TYPE`* `to` *`TYPE`* `is illegal.`	Compiler Error	ANSI C does not allow a cast between the two types.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`COFF Name too long.`	Object Load Error	The COFF object or library you are loading contains a symbol name that is longer than the maximum legal length.
`Comparison involving null pointer.`	Non-Fatal Runtime 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 Runtime Error	One of the pointer expressions in the comparison is invalid because it was not initialized.
`Comparison of pointers to different objects.`	Non-Fatal Runtime Error	The 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 Runtime Error	One of the pointer expressions in the comparison is invalid because it refers to a location in dynamic memory that was deallocated with the function `free.` Once memory is freed, all pointers into that block of memory become invalid.
`Compound statements nested too deeply.`	Compile Error	The program has exceeded the compiler limitations on the number of nested compound statements; reduce the depth of the nested compound statements in the program.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Conditional inclusion nested too deeply.`	Compile Error	The 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	The object module is probably corrupted.
`Conflicting argument declarations for function` *`FUNCTION`*`.`	Compile Error	The 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	A 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	Converting between a function pointer and other types of pointers is discouraged because functions should not be accessed as data and data cannot be executed as functions.
`Could not allocate stack space. Try decreasing the Maximum stack size option in the Run Options dialog.`	Fatal Runtime Error	There is insufficient memory to allocate the Maximum Stack Size you have specified. LabWindows/CVI allocates the maximum size on the stack at the beginning of execution.

(continues)

Table 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 containing the prototypes for the functions in the DLL. When generating glue code, Ensure that correct file name is specified. 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 Runtime Error	The `d` modifier cannot be used in `Fmt`, `FmtOut`, or `FmtFile`.
`Declaration of` *`NAME`* `does not match previous declaration at` *`POSITION`*`.`		A variable or function has been declared twice, and its type in the first declaration does not match its type in the second declaration.
`Declared parameter` *`NAME`* `is missing.`	Compile Error	A parameter declaration to an old-style function definition is missing or was not declared in the function parameter 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	The preprocessor `defined()` operation requires a single identifier argument; Ensure that an identifier is being used and not an expression.
`Dereference of a` *`NUMBER`* `byte object where only` *`NUMBER`* `bytes exist.`	Fatal Runtime Error	The pointer expression being 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`, the pointer cannot be dereferenced because it points to only one byte, whereas four bytes are required for an `int`.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Dereference of data pointer used as a function.`	Fatal Runtime Error	A pointer to data was converted to a function pointer and dereferenced. Data can only be examined or modified, and cannot be executed as a function.
`Dereference of function pointer used as data.`	Fatal Runtime Error	A function pointer was converted to a non-function pointer and dereferenced. Functions can only be executed, and cannot be accessed as data.
`Dereference of invalid pointer expression.`	Fatal Runtime Error	The pointer expression being dereferenced is invalid. It is probably the result of a previous invalid pointer operation.
`Dereference of null pointer.`	Fatal Runtime Error	The pointer expression being dereferenced has the value `NULL` and cannot be dereferenced.
`Dereference of out-of-bounds pointer:` *`NUMBER`* `bytes (`*`NUMBER`* `elements) before start of array.`	Fatal Runtime Error	The pointer expression being dereferenced is invalid because it refers to a location that is before the start of an array. The number of bytes and the number of array elements in the array is given. 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 Runtime Error	The pointer expression being dereferenced is invalid because it refers to a location that is past the end of an array. The number of bytes and the number of array elements past the end of the array is given. The expression is probably the result of previous illegal pointer arithmetic.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Dereference of pointer to freed memory.`	Fatal Runtime Error	The pointer expression being dereferenced is invalid because it refers to a location in dynamic memory that was deallocated with the function `free`. Once memory is freed, all pointers into that block of memory become invalid.
`Dereference of unaligned pointer.`	Fatal Runtime Error [UNIX only]	The pointer expression being 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, that 32-bit objects be word aligned, and that 64-bit objects be doubleword aligned.
`Dereference of uninitialized pointer.`	Fatal Runtime Error	The pointer expression being dereferenced is invalid because it was not initialized. It is probably an uninitialized local variable. Local variables must be initialized before they are used.
`Duplicate case label` *`NAME`*`.`	Compile Error	A 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	A previously defined parameter name has been redeclared; eliminate one of the parameter declarations.
`Duplicate field name` *`NAME`* `in` *`TYPE`*`.`	Compile Error	The member name of the `struct` or `union` type has already been declared. Eliminate one of the member declarations from the struct or union type declaration.
`Dynamic memory is corrupt.`	Fatal Runtime Error	LabWindows/CVI encountered corrupt data while allocating or freeing dynamic memory.
`Empty declaration.`	Compile Error or Warning	No object or type is declared. It is an error if the empty declaration appears in the context of an old-style parameter declaration.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Elf library is out of date.`	Object Load Error	CVI expects a more recent version of the shared library (`libelf.so`) that it uses to load ELF objects. As a result, LabWindows/CVI is unable to read or write object and library files.
`'enum` *`NAME`*`' declared inside parameter list has scope only for this declaration.`	Compile Warning	The 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 should declare the enumeration type before declaring function types that use it.
`Error at or near character` *`NUMBER`* `in the format string:` *`STRING`*`.`	Non-Fatal Runtime Error	There is an error in the format string at index *`NUMBER`*. *`NUMBER`* is 1 based.
`Error in Elf Library encountered while reading external module: NAME.`	Object Load Error	The object module is corrupted or is of a type that cannot be loaded by LabWindows/CVI.
`Error: compiling` *`FILE`* `for DLL exports.`	DLL Import Library Creation Error.	When creating a DLL using the Include File method for specifying exported symbols, an error was encountered compiling the include file.
`Error: Incompatible type for function or variable` *`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 does not match the type in the source file.
`Expecting an enumerator identifier.`	Compile Error	An enumeration constant identifier was expected after the opening { in an `enum` type declaration.
`Expecting an identifier.`	Compile Error	An identifier was expected in the current syntactic context. Check the syntax of the declaration, statement, or preprocessor directive.
`Expecting integer constant, push or pop.`	Compile Error	The pack pragma requires at least one parameter.
`Extra default label.`	Compile Error	A `default` label has already appeared for this switch statement. Eliminate the extraneous `default` label.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Extraneous 0-width bit field` *`TYPE NAME`* `ignored.`	Compile Warning	The named bit field has no width and therefore has no storage allocated to it.
`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 , if this is an old-style function declaration, the parameter names only appear in the function definition and not in any declaration of that function. If this was intended to be a new-style function declaration (prototype), then probably the identifier that is assumed to be a parameter name by the compiler probably was intended to be a `typedef` name. Make sure that the identifier has been previously declared as a `typedef`.
`Extraneous identifier` *`NAME`*`.`	Compile Error	An identifier appears in a context where a type name is expected, such as in a cast operation or as the operand of `sizeof()`. A type name is syntactically a declaration for a function or an object of that type that omits the identifier.
`Extraneous return value.`	Compile Error	The return statement appears in a void function and therefore no return value is necessary; eliminate the return 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	An external module could not be opened for loading. Check that the external module has read access and that it has not inadvertently been renamed or deleted.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Field name expected.`	Compile Error	An identifier is syntactically expected following a `.` or `->`.
`Field name missing.`	Compile Error	The identifier is missing from a member (field) declaration in a `struct` or `union` type declaration. Make sure that the member type specifier is followed by an identifier.
`Format string integer is too big.`	Non-Fatal Runtime Error	An integer used in the format string is too large.
`Found` *`TYPE`* `expected a function.`	Compile Error or Warning	In an C expression, the name of a function or pointer to function was expected to precede the `(`. In a `#pragma` line, the name of a function was expected after the `pragma` type.
`Function definitions are not allowed in the interactive window.`	Compile Error	A function definition cannot appear in the interactive window; move the function definition to a program window and call it from the interactive window.
`Function` *`FUNCTION`*`: (`*`STRING`* `==` *`NUMBER`*`).`	Non-Fatal Runtime Error	The 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. See the library function reference material for more information about the error.
`Function` *`FUNCTION`* `has an unsupported return type size.`	Glue Code Generation Error	The return type of the function is not supported by the glue code generation, or the DLL loading facilities.
`Function requires extra code to handle Callbacks:` *`FUNCTION`*`.`	Glue Code Generation Error	The 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. Such functions need to be edited to add the code to `thunk` function pointers.

(continues)

Table A-1. Error Messages (Continued

Error Message	Type Error	Comment
h modifier is only valid with d, i, n, o, u, and x specifiers.	Non-Fatal Runtime Error	The h modifier can only be used with integer format specifiers.
Header name literal too long.	Compile Error	Header name length exceeds implementation limitations. Check that header name is properly terminated with a > or a ", or shorten string literal.
Ill-formed constant integer expression.	Compile Error	A constant integer expression appearing in a preprocessor directive is syntactically invalid. Check the expression for trailing tokens.
Ill-formed hexadecimal escape sequence \x*CHAR*.	Compile Error	Check that a hexadecimal character ([0-9a-fA-F]) follows the \x escape sequence introduction.
Ill-formed hexadecimal escape sequence.	Compile Error	Check that a hexadecimal character ([0-9a-fA-F]) follows the \x escape sequence introduction.
Illegal argument(s) to library function.	Fatal Runtime 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	A case label appears outside the context of a switch statement. Remove the case label.
Illegal character *CHAR*.	Compile Error	A character or character escape sequence that is outside the set of the legal character set for an ANSI C file appears in a context other than a character string or character literal.
Illegal continue statement.	Compile Error	A continue statement appears outside a loop statement. Remove the continue statement.
Illegal default label.	Compile Error	A default label appears outside the context of a switch statement. Remove the default label.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
Illegal expression.	Compile Error	A token was encountered while parsing an expression where an identifier, string literal, integer constant, floating constant, or (was expected.
Illegal extern definition of *NAME*; all interactive window variable definitions must be static.	Compile Error	No interactive window definitions may be visible outside the scope of the Interactive window. You cannot initialize external objects in the Interactive window.
Illegal formal parameter types.	Compile Error	A 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	An unexpected character follows an #include where a header file name of the form "FILE" or <*FILE*> is expected. It is also possible that the header file name 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.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
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	An illegal attempt to initialize an extern declaration that appears in a local scope was made. Eliminate the initialization.
Illegal return type *TYPE*.	Compile Error	A function was 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, check that the return type is not an array type or a function type. If the diagnostic is for a return statement, the containing function was probably declared void and requires no expression for a return statement.
Illegal return type; found *TYPE* expected *TYPE*.	Compile Error	A return statement expression type is not the same as the return type of the function in which it appears. Check 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 Runtime 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	The only token expected to follow the line number specification in a #line preprocessor directive is an optional string literal specifying a source file name. Alternatively, a sequence of tokens may follow the #line token as long as once macro expansion on the containing source line is performed it conforms to one of the two allowable forms of #line preprocessor directives: #line *line-number-digit-sequence* #line *line-number-digit-sequence "filename"*

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Illegal statement termination.`	Compile Error	During compilation of a sequence of statements, a token was encountered that was 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 diagnostic was issued, check that the statement syntax is correct for the cases listed above.
`Illegal type const` *`TYPE`*`.`	Compile Error	A qualified type specification (that is, one containing the keyword `const` or `volatile`) is specified with the same qualifier more than once (such as, `const const int`). Ensure that the `const` and `volatile` qualifiers are not used more than once each in the same type.
`Illegal type for symbol 'DllMain': TYPE.`	Compile Error	The function `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	The function `WinMain` does not conform to the accepted prototype. `int__stdcall WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow);`
`Illegal type array of` *`TYPE`*`.`	Compile Error	An attempt to declare an array of function type was made. Probably the declaration was intended to be an array of pointer to function type instead.
`Illegal type volatile` *`TYPE`*`.`	Compile Error	A qualified type specification (that is, one containing the keyword `const` or `volatile`) is being specified with the same qualifier more than once (such as `const const int`). Ensure that the `const` and `volatile` qualifiers are not used more than once each in the same type.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Illegal use of type name` *`NAME`*`.`	Compile Error	A `typedef` name was used in the context of a primary expression. If a type cast was intended, ensure that the `typedef` name is parenthesized. Otherwise a macro name, enumeration constant, variable name, or function name was intended.
`Illegal value matched to asterisk.`	Non-Fatal Runtime Error	An integer argument matched to an asterisk (`*`) in the format string has an invalid value given the context in which it appears.
`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	A global variable marked as `__import` or `declspec(dllimport)` is being used in an initializer of another variable.
`Include files nested too deeply.`	Compile Error	The 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	The current declaration of the identifier is inconsistent with a previous declaration of the same identifier with regard to linkage. Check that all declarations of the identifier that are intended 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	The types declared for two or more external symbols of the name are not the same. Check each program file containing an external declaration of the symbol for type consistency.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Initializer exceeds bit-field width.`	Compile Warning	The number of bits needed to represent the initialization value of a bit-field exceeds its declared width. The initialization value is truncated to fit the bit-field. The initialization value should be smaller or the bit-field declaration made wider.
`Initializer must be constant.`	Compile Error	The initializer must be an expression that conforms to the semantics for a constant expression.
`InitVXIlibrary must be called before other VXI functions.`	Fatal Runtime Error	The function `InitVXIlibrary` must be called before other VXI functions.
`Insufficient number of arguments to` *`FUNCTION`*`.`	Compile Error	The function expects more arguments than it is being called with. Check the function declaration for the number of parameters declared.
`Insufficient system memory for Interactive Window`	Link Error	There is not enough memory to run the interactive window.
`Insufficient system memory for project.`	Link Error	There is not enough memory to link the project.
`Insufficient user data memory for project.`	Link Error	There is not enough memory to link the project.
`Invalid hexadecimal constant.`	Compile Error	A token assumed to be a hexadecimal constant is badly formed. Check that token conforms to syntax for hexadecimal constants, especially that a valid hexadecimal digit follows the `0x` or `0X` prefix.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Invalid initialization type; found` *`TYPE`* `expected` *`TYPE`*`.`	Compile Error	The expression initializing the object declaration is type incompatible with the object. Check 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 any aggregate type, such as member types of a `struct` or `union` type.
`Invalid octal constant.`	Compile Error	A token assumed to be an octal constant is badly formed. Check that token conforms to syntax for octal constants especially that a valid octal digits follow the leading `0` prefix.
`Invalid operand of unary &;` *`NAME`* `is declared register.`	Compile Error	It is illegal for the address to be taken (`&` prefix operator) of an object declared to be of `register` class. Remove the `register` keyword from the object declaration if the address operator will be applied to it.
`Invalid pointer argument to library function.`	Fatal Runtime Error	The pointer expression being passed to the library function is invalid. It is probably the result of a previous invalid pointer operation.
`Invalid size for a real.`	Non-Fatal Runtime Error	The only valid sizes (specified with the `b` modifier) for the `f` (real) specifier are 4 and 8.
`Invalid size for an integer.`	Non-Fatal Runtime Error	The valid sizes (specified with the `b` modifier) for the `i`, `d`, `x`, `o`, and `c` modifiers are 1 to 4.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Invalid storage class.`	Compile Error	The only allowable explicit storage class specifier for a function declaration that has block scope is `extern`.
`Invalid struct field declarations.`	Compile Error	An invalid token was encountered while processing a struct or union type declaration. A token beginning a member type specifier was expected where a 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	The `sizeof` operator was applied 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 not known before its full declaration.
`Invalid type specification.`	Compile Error	The combination of type specifiers is incompatible. The type specifier `short` may only be used in combination with `int`. The type specifier `long` may only be used in combination with `int` and `double`. The type specifiers `signed` and `unsigned` may only be used in combination with one of the basic integer type (`char`, `short`, `int`, `long`).

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Invalid union field declarations.`	Compile Error	An invalid token was encountered while processing a struct or union type declaration. A token beginning a member type specifier was expected where a 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 Runtime Error	The `l` format specifier is only used in `Scan`, `ScanOut`, and `ScanFile`.
`l modifier is only valid with d, i, n, o, u, and x specifiers.`	Non-Fatal Runtime Error	The `l` format specifier is only valid for integer format specifiers.
`L modifier is only valid with e, f, and g specifiers.`	Non-Fatal Runtime Error	The `L` modifier, which specifies that the argument is a long double, can only be used in the floating point formats.
`l modifier is only valid with e, f, g, d, i, n, o, u, and x specifiers.`	Non-Fatal Runtime Error	The `l` format specifier is only valid for integer and real format specifiers.
`Left operand of -> has incompatible type` *TYPE*.	Compile Error	The 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.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Left operand of . has incompatible type` *`TYPE`*`.`	Compile Error	The left operand of the `.` member selection operation must be a `struct` or `union` type.
`Library function error (`*`STRING`*` == `*`NUMBER`*`).`	Non-Fatal Runtime Error	The 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. See 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	The macro expansion has exceeded the compiler implementation size limitation.
`Macro parameter must follow # operator.`	Compile Error	The # operator requires that a macro parameter immediately follow it in a macro replacement list.
`Matching push not encountered or already popped.`	Compile Error	A pack pragma used a named pop that does not balance with push of the same name.
`Missing { in initialization of` *`TYPE`*`.`	Compile Error	The initialization of a `struct`, `union`, or `array` type, is missing a starting { for an aggregate initialization value.
`Missing #endif`	Compile Error	The `#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 file name follows the `#include` preprocessor directive. Ensure that there is a filename of the correct form following `#include` or that any macro following `#include` expands into the correct form for an include file name.
`Missing '.`	Compile Error	The termination single quote character `'` is missing from a character or wide character literal.
`Missing` *`CHAR`*`.`	Compile Error	Check for unterminated string or character literal.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Missing argument to variable argument function.`	Fatal Runtime Error	The variable argument function required at least one argument beyond the last formal parameter.
`Missing array size.`	Compile Error	Attempting 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 specified size.
`Missing format string integer.`	Non-Fatal Runtime Error	The integer corresponding to an asterisk in a format string is missing. This may be caused by incorrect ordering of the arguments. This integer must precede the actual argument.
`Missing identifier.`	Compile Error	An identifier specifying the object name is missing from the object declaration. Ensure that the object type specifier is followed by an identifier.
`Missing label in goto.`	Compile Error	The `goto` statement is missing an identifier label.
`Missing parameter name to function` *`FUNCTION`*`.`	Compile Error	The 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 should not be an identifier.
`Missing parameter type.`	Compile Error	The 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.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Missing prototype.`	Compile Error	The function declaration or call is for a function without prototype declaration information. The diagnostic is issued as indicated by the **Require Function Prototypes** compiler option.
`Missing return value.`	Fatal Runtime Error	The function does not return a value, although it was declared with a return type. If the function was not intended to return a value, then it should be declared as a `void` function. Otherwise, it must have a `return` statement to return a value.
`Missing return value.`	Compile Warning	The non-void function does not return a value. Add a return statement with an expression of the function return type. This diagnostic is issued as under the control of the **Require return value for non-void functions** compiler preference.
`Missing right bracket (]).`	Fatal Runtime Error	The format string has mismatched brackets.
`Missing struct tag.`	Compile Error	A tag name is missing from an incomplete struct or union declaration.
`Missing terminating null in string argument.`	Fatal Runtime Error	The library function expects a string argument, but the passed argument points to an array of characters that is not null terminated.
`Missing union tag.`	Compile Error	A tag name is missing from an incomplete struct or union declaration.
`Missing Watch Expression.`	Watchpoint Error	The watch expression was not specified in the watch dialog box.
`Multiply defined symbol` *`NAME`* `in modules` *`FILE1`* `and` *`FILE2`*`.`	Link Error	There exists more that one definition for *`NAME`* among the elements of the project being linked.
`naked functions are not supported.`	Compile Error	LabWindows/CVI does not work with the `naked` keyword.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
No data relocation section found for external module: *FILE*.	Link Error	The external object module does not contain the relocation information necessary to link it in with the rest of the project. Ensure that the external object module being loaded is not an already linked executable.
No data section found for external module: *FILE*.	Link Error	The external object module does not contain the initialized data necessary to link it in with the rest of the project. Check how the external object file was built.
No pack settings currently pushed.	Compile Error	A pack pragma used a pop when there were no pushes.
No symbol table found for external module: *FILE*.	Link Error	The external object module does not contain the symbol table information necessary to link it in with the rest of the project. Check how the external object file was built.
No text relocation section found for external module: *FILE*.	Link Error	The external object module does not contain the relocation information necessary to link it in with the rest of the project. Ensure that the external object module being loaded is not an already linked executable.
No text section found for external module: *FILE*.	Link Error	The external object module does not contain the initialized instruction data necessary to link it in with the rest of the project. Check how the external object file was built.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Non-terminated address list.`	Fatal Runtime Error	An attempt was made to pass an address list array not terminated with a -1 to a GPIB-488.2 function that expects the array to be terminated with a -1.
`Not enough parameters.`	Non-Fatal Runtime Error	The number of arguments expected by the format string is more than the number of arguments passed in.
`Not enough space for casting expression to` *`TYPE`*`.`	Non-Fatal Runtime Error	The block of memory obtained from `malloc` or `calloc` is not large enough for a single object of type *`TYPE`* and cannot be cast to that type.
`Null Pointer.`	Fatal Runtime Error	The pointer expression being 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 Runtime Error	The pointer expression being passed to the library function has the value `NULL`, which is not a valid value for the function.
`Number of arguments exceed the maximum supported.`	Non-Fatal Runtime Error	The number of arguments exceeds the maximum supported by the formatting functions.
`Number of points is too large for current waveform buffer.`	Fatal Runtime Error	This 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.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Object module contains unsupported FAR pointers.`	Object Load Error	The 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	The type of one of the function's arguments is not supported by the glue code generation, or the DLL loading facilities.
`Only object modules produced by WATCOM C 386 fully supported.`	Link Error	The external object module contains unrecognized or unsupported OMF records. Ensure that the object file was compiled with a WATCOM C 386 compiler with the recommended options.
`Operands of '=' have incompatible calling conventions.`	Compile Error	A function pointer is being 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	The 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	The types of the operands to the binary operator are not compatible according to the ANSI C standard.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Operand of unary` *`OPERATOR`* `has illegal type` *`TYPE`*`.`	Compile Error	The type of the operand to the unary operator is not valid.
`Out-of-bounds pointer argument (before start of array).`	Fatal Runtime Error	The pointer expression being 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 Runtime Error	The pointer expression being passed to the library function is invalid because it refers to a location that is 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 Runtime Error	The pointer arithmetic expression is invalid because the resulting value refers to a location that is before the start of an array. The number of bytes and number of array elements past the end are given.
`Out-of-bounds pointer arithmetic:` *`NUMBER`* `bytes (`*`NUMBER`* `elements) past end of array.`	Non-Fatal Runtime Error	The pointer arithmetic expression is invalid because the resulting value refers to a location that is past the end of an array. The number of bytes and number of array elements past the end are given.
`Out of memory for user protection information.`	Fatal Runtime Error	Could not allocate memory required to store user protection information.
`Overflow in constant` *`CONSTANT`*`.`	Compile Warning	The value of a constant or constant expression exceeds the environmental limits of the type. Check that the value will not exceed the maximum value for the expression type.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Overflow in constant expression.`	Compile Warning	The value of a constant or constant expression exceeds the environmental limits of the type. Check that the value will not exceed the maximum value for the expression type.
`Overflow in floating constant` *`CONSTANT`*`.`	Compile Warning	The value of a constant or constant expression exceeds the environmental limits of the type. Check that the value will not exceed the maximum value for the expression type.
`Overflow in hexadecimal escape sequence.`	Compile Warning	The value of a constant or constant expression exceeds the environmental limits of the type. Check that the value will not exceed the maximum value for the expression type.
`Overflow in octal escape sequence.`	Compile Warning	The value of a constant or constant expression exceeds the environmental limits of the type. Check that the value will not exceed the maximum value for the expression type.
`Overflow in value for enumeration constant` *`CONSTANT`*`.`	Compile Error	The value of a constant or constant expression exceeds the environmental limits of the type. Check that the value will 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 Runtime Error	The number was too large to be stored 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 Runtime Error	The number was too large to be stored in a 4 byte real.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Overflow occurred during the conversion of the int. The signed value is too big for the size.`	Non-Fatal Runtime Error	The number was too large to be stored in the integer of the specified size.
`Pack pragma valid values are 1, 2, 4, 8, and 16.`	Compile Error	The pack pragma alignment value parameter is limited to 1, 2, 4, 8, or 16.
`Parameter type incompatible with format specifier.`	Non-Fatal Runtime Error	The parameter type is not compatible with the type expected by the format string. An argument is either missing or of the wrong type.
`Parameter type mismatch: expecting` *`TYPE`* `but found` *`TYPE`*`.`	Non-Fatal Runtime Error	The parameter type does not match the type expected by the format string. The arguments may not be in the right order, or an argument may have been omitted.
`Pointer arithmetic involving invalid pointer.`	Non-Fatal Runtime Error	The pointer arithmetic expression is invalid because one of the subexpressions contains an invalid pointer.
`Pointer arithmetic involving null pointer.`	Non-Fatal Runtime Error	The pointer arithmetic expression is invalid because one of the subexpressions contains a the value `NULL`.
`Pointer arithmetic involving pointer to freed memory.`	Non-Fatal Runtime Error	The pointer arithmetic expression is invalid because one of the subexpressions contains a pointer to dynamic memory that was deallocated with the function `free`. Once memory is freed, all pointers into that block of memory become invalid.
`Pointer arithmetic involving pointer to function.`	Non-Fatal Runtime Error	The pointer arithmetic expression is invalid because one of the subexpressions is a function pointer.
`Pointer arithmetic involving uninitialized pointer.`	Non-Fatal Runtime Error	The pointer arithmetic expression is invalid because one of the subexpressions contains an uninitialized pointer. It is probably an uninitialized local variable.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Pointer comparison involving address of nonarray object.`	Non-Fatal Runtime 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.
`Pointer is invalid.`	Non-Fatal Runtime Error	A pointer argument to the function contains an invalid address.
`Pointer points to freed memory.`	Non-Fatal Runtime Error	A pointer argument to the function points to memory that has been freed.
`Pointer subtraction involving address of nonarray object.`	Non-Fatal Runtime 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	The value being returned from the function is a pointer to a parameter or local variable. Because the lifetime of a parameter or local ends upon function return, any pointer to such an object is invalid and therefore in error.
`Pointer to a parameter is an illegal return value.`	Compile Error	The value being returned from the function is a pointer to a parameter or local variable. Because the lifetime of a parameter or local ends upon function return, any pointer to such an object is invalid and therefore in error.
`Pointer to free memory passed to library function.`	Fatal Runtime Error	The pointer expression being passed to the library function is invalid because it refers to a location in dynamic memory that was deallocated with the function `free`. Once memory is freed, all pointers into that block of memory become invalid.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`pragma pack(pop...) does not set alignment. Use separate pack pragma.`	Compile Warning	A pragma pop was used with an alignment value. Use separate pack pragmas for popping and setting the alignment value.
`Project not linked.`	Link Warning	This error occurs when one or more link errors are reported in the Interactive Execution window and the project is not in a linked state, and provides a possible explanation for the link errors. The Interactive Execution window cannot be linked 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.
`Qualified function type ignored.`	Compile Warning	Any qualification of a function declaration is extraneous but harmless.
`Read error.`	Link Error	An error has occurred while attempting to read a file. Check 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	A function has been redeclared with a different calling convention.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Redeclaration of macro parameter` *`NAME`*`.`	Compile Error	The parameter name has already been used once by the macro. Choose another parameter name.
`Redeclaration of` *`NAME`*`.`	Compile Error	The declared name conflicts with a previous declaration in the same scope and name space. The name has already been used in this scope. Choose another for this declaration.
`Redeclaration of` *`NAME`* `previously declared at` *`POSITION`*`.`	Compile Error	The declared name conflicts with a previous declaration in the same scope and name space. The name has already been used in this scope; choose another for this declaration.
`Redefinition of label` *`NAME`* `previously defined at` *`POSITION`*`.`	Compile Error	The statement label has already been used in this function. A statement label must be unique within the function in which it used.
`Redefinition of macro` *`NAME`*`.`	Compile Error	The 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	The object or function has already been defined in the current scope. Eliminate one of the two definitions.
`Reference parameter expected.`	Non-Fatal Runtime Error	The function expected a pointer but was passed a scalar.
`Register declaration ignored for` *`TYPE NAME`*`.`	Compile Warning	The `register` storage class conflicts with the semantics of the type declared for the object. If the object has been declared to be of an array or `struct` or `union` type or is qualified to be `volatile` then remove the `register` keyword from the declaration.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Register declaration ignored for` *`TYPE`* `parameter.`	Compile Warning	The `register` storage class conflicts with the semantics of the type declared for the parameter prototype. If the object has been declared to be of `struct` or `union` type or is qualified to be `volatile` then remove the `register` keyword from the prototype parameter declaration.
`Repeat value not valid with s/l format specifiers.`	Non-Fatal Runtime Error	A repeat value cannot be used with the `s` and `l` format specifiers.
`Result of unsigned comparison is constant.`	Compile Warning	The result of <*`UNSIGNED INTEGER EXPRESSION`*> `>= 0` will always evaluate to 1.
`Segment must be of class CODE, DATA, BSS, or STACK: segment name` *`NAME`*`.`	Load Error	The external object module contains an unknown class of segment. Object modules should not contain any specially named segments.
`Segment must be USE32: segment name` *`NAME`*`.`	Link Error	The external object module being loaded contains unsupported 16 bit segments. LabWindows/CVI only supports 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 that may be the source of problems if the `lvalue` is later dereferenced. This diagnostic is issued if you select the **Enable signed/unsigned pointer mismatch warning** compiler option.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Simple/Array conflict with format specifier.`	Non-Fatal Runtime Error	An array passed to the function is matched to a format specifier for a scalar, or a scalar passed to the function is matched to a format specifier for an array.
`Size of array of` *`TYPE`* `exceeds` *`SIZE`* `bytes.`	Compile Error	The size of the array or struct/union type exceeds the compiler limitation of `INT_MAX` bytes.
`Size of` *`TYPE`* `exceeds` *`SIZE`* `bytes.`	Compile Error	The size of the array or struct/union type exceeds the compiler limitation of `INT_MAX` bytes.
`sizeof applied to a bit field.`	Compile Error	Ensure that the `sizeof()` operation is not being used on a bit field.
`Specified width is too small to read the number.`	Non-Fatal Runtime Error	The width specified for a format specifier was not large enough to read in a complete number. Example: a width of `2` specified for a float, and the number is `-.02`; the negative sign and decimal point do not make a valid number.
`Stack Overflow.`	Fatal Runtime Error	The program exceeded the stack limit. Change the size of the stack in the run options, 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	The 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 should declare the structure type before declaring function types that use it.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`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 Runtime 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 Runtime 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 Runtime Error	One of the pointer expressions in the subtraction is invalid because it was not initialized.
`Subtraction of pointers to different objects.`	Non-Fatal Runtime Error	The pointer expressions in the subtraction point to two distinct objects. Both expressions in pointer subtractions must point into the same array object.
`Subtraction of pointers to freed memory.`	Non-Fatal Runtime Error	One of the pointer expressions in the subtraction is invalid because it refers to a location in dynamic memory that was deallocated with the function `free`. Once memory is freed, all pointers into that block of memory become invalid.
`Switch statement with no cases.`	Compile Warning	The switch statement contains no `case` or `default` label.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`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 creating a DLL using the Include File method for specifying exported symbols, one of the symbols declared in the include file was not found in the DLL project. Or, when creating import libraries from an include file and a DLL, one of the symbols declared in the include file was not found in the DLL.
`Syntax error; found TOKEN1 expecting TOKEN2.`	Compile Error	A syntax error occurred because TOKEN1 was found instead of TOKEN2.
`The __cdecl calling convention is not supported with functions returning floats, doubles, or structures in WATCOM Compatibility Mode.`	Compile Error	A function with an explicit `__cdecl` qualifier returns a `double`, `float` or structure, and the selected compatible compiler is WATCOM. Either remove the qualifier or change the function.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`The callback function, NAME, differs only by a leading underscore from another function or variable. Change one of the names for proper linking.`	Non-Fatal Runtime Error	When trying to match a callback name specified in a `.uir` file to the callback function, the compiler found two symbols that are the same except for a leading underscore. Resolve this ambiguity by changing the one of the names.
`Thread data is not supported.`	Compile Error	You cannot implement thread-local storage in LabWindows/CVI.
`Too many arguments to` *`FUNCTION`*`.`	Compile Error	The declaration for function *`FUNCTION`* contains fewer parameters than the number of arguments passed in this function call.
`Too many arguments to variable argument function.`	Non-Fatal Runtime Error	More arguments were passed to the variable argument function than were expected. The extra arguments will not affect the function call in any way.
`Too many function parameters.`	Compile Error	The number of parameter declarations exceeds compiler limitations. Declare the function with fewer parameters.
`Too many initializers.`	Compile Error	The 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	The number of parameter declarations exceeds compiler limitations. Declare the macro with fewer parameters.
`Too many parameters.`	Non-Fatal Runtime Error	The number of parameters passed to a function exceed the number of parameters expected by the format string.
`Type error in argument %d to %s, calling convention mismatch.`	Compile Error	The function or function pointer being passed to a function does not have the correct calling convention.
`Type error in argument` *`NUMBER`* `to` *`NAME`*`;` *`TYPE`* `is illegal.`	Compile Error	The argument being passed is an illegal array type or an incomplete type of which the size is not known. Ensure that the argument is of a complete type.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Type error in argument` *`NUMBER`* `to` *`NAME`*`; found` *`TYPE`* `expected` *`TYPE`*`.`	Compile Error	An argument that is not type compatible with the prototype declaration for the parameter in that position has been passed. Ensure that the actual argument is type compatible with the parameter declaration.
`Type error: pointer expected.`	Compile Error	The expression being dereferenced with the '`*`', '`->`' or '`[ ]`' operator does not have pointer type.
`Unclosed comment.`	Compile Error	A comment is missing the closing `*/` delimiter.
`Undeclared identifier` *`NAME`*`.`	Compile Error	*`NAME`* has not been previously declared. All names must be declared before use. Check that *`NAME`* declaration has not been conditionally excluded from compilation.
`Undefined label` *`NAME`*`.`	Compile Error	The label *`NAME`* is used as the target of at `goto` statement in the function but 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 function `goto` are illegal.
`Undefined size for` *`TYPE NAME`*`.`	Compile Error	An object has been defined 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	A member (field) declaration has no size for the declared type. Member was probably declared with an empty `struct` or `union` type declaration.
`Undefined size for field` *`TYPE NAME`*`.`	Compile Error	A member (field) declaration has no size for the declared type. Member was probably declared with an empty `struct` or `union` type declaration.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Undefined size for parameter` *`TYPE NAME`*`.`	Compile Error	An parameter has been declared 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	A `static` object has been declared with an incomplete type or no initialization expression from which a size for the type may be surmised. Because the size of an incomplete type is unknown, storage cannot be allocated for the object.
`Undefined static` *`TYPE NAME`*`.`	Compile Warning or Error	The `static` function was declared but never defined. Because a `static` function is only visible within the file it was declared, it must be defined at some point within the file in order to be used. If the function was called anywhere in the file this diagnostic is an error. Otherwise it is a warning.
`Undefined symbol` *`NAME`*`.`	Link Error	A variable or function is used in the project, but is not defined anywhere.
`Unexpected #elif; #endif expected.`	Compile Error	A `#else` preprocessor directive was encountered immediately prior to this `#elif` at the same level of conditional inclusion. Check that the conditional preprocessor inclusion directives at this level are in the proper order.
`Unexpected #elif; #if not seen.`	Compile Error	An `#elif` preprocessor directive is encountered where no previous beginning `#if`, `#ifdef`, or `ifndef` was seen at this level.
`Unexpected #else; #endif expected.`	Compile Error	An `#else` preprocessor directive was encountered immediately following a prior `#else` at the same level of conditional inclusion. Check that the conditional preprocessor inclusion directives at this level are in the proper order.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Unexpected #else; #if not seen.`	Compile Error	An `#else` preprocessor directive is encountered where no previous beginning `#if`, `#ifdef`, or `ifndef` was seen at this level.
`Unexpected #endif; no matching #if, #ifdef, or #ifndef.`	Compile Error	An `#endif` preprocessor directive is encountered where no previous beginning `#if`, `#ifdef`, or `ifndef` was seen at this level.
`Unexpected EOF.`	Load Error	An unexpected End Of File (EOF) condition was encountered when loading an external object module. Check that the object file has not been truncated.
`Unexpected EOF;` *`TOKEN`* `expected.`	Compile Error	An End Of File (EOF) condition was encountered while in mid-parsing of a syntactic construct. Check that syntactic structure are complete, for example, matching parenthesis, matching braces).
`Unexpected end of format string.`	Non-Fatal Runtime Error	The format string 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	An unexpected token was encountered while processing a `#define` preprocessor directive. Check for missing `)` in any macro parameter list
`Unexpected trailing tokens on directive line ignored`	Compile Warning	Preprocessor line contains harmless trailing tokens that are ignored.
`Uninitialized pointer.`	Non-Fatal Runtime Error	A pointer argument passed to a function is uninitialized.
`Uninitialized pointer argument to library function.`	Fatal Runtime Error	The pointer expression being passed to the library function is invalid because it was not initialized. It is either a local variable or an object in dynamic memory that was not initialized.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`Uninitialized string.`	Non-Fatal Runtime Error	The pointer argument passed to the library function has not yet been initialized, or is `NULL`.
`'union` *`NAME'`* `declared inside parameter list has scope only for this declaration.`	Compile Warning	The 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 declaring function types that use it.
`Unknown enumeration` *`NAME`*`.`	Compile Error	*NAME* is an undeclared enumeration type.
`Unknown field` *`NAME`* `of` *`TYPE`*`.`	Compile Error	A 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 being selected or dereferenced.
`Unknown modifier.`	Non-Fatal Runtime 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	An unknown OMF record was encountered while loading an external object module. Ensure that the external object module was compiled properly.
`Unknown size of type` *`TYPE`*`.`	Compile Error	Pointer arithmetic is being performed on operand(s) that are pointers to types of unknown size which are probably incomplete types or pointer to function types. Ensure that the pointer types are to fully declared types and that the pointer types are not pointers to functions either.

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment
`Unknown specifier.`	Non-Fatal Runtime Error	The specifier in the format specifier is not valid.
`Unnamed pop matching named push.`	Compile Warning	A pack pragma used a unnamed pop that balances a name push.
`Unrecognized character escape sequence.`	Compile Warning	The 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	The character escape sequence does not conform to any known character escape sequence, octal escape sequence, or hexadecimal escape sequence.
`Unrecognized declaration.`	Compile Error	The declaration is unrecognizable. Check declaration syntax of the function, object, or type that is being declared.
`Unrecognized preprocessor directive.`	Compile Error	A # character begins a preprocessor directive that is unknown. Check the spelling of the preprocessor directive.
`Unrecognized statement.`	Compile Error	The statement syntax is unrecognizable. Check statement syntax of type of statement that was intended.
`Unsigned operand of unary -.`	Compile Warning	A nonsensical unary – operation is being performed on an unsigned type. A negation operation on an unsigned type is not effective.
`Unsupported segment combination type` *`NUMBER`*`: segment name` *`NAME`*`.`	Load Error	A bad segment was encountered 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	The use of a keyword in a variable definition contradicts a previous definition, e.g.
		`int __export x;`
		`int __import x=0;`

(continues)

Table A-1. Error Messages (Continued)

Error Message	Type Error	Comment

`Use of keyword '__export' contradicts previous use of keyword '__import' at` *POSITION*.	Compile Error	The use of a keyword in a variable definition contradicts a previous definition, e.g.
		`int __import x;`
		`int __export x=0;`

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

`Value parameter expected.`	Non-Fatal Runtime Error	A pointer was passed for a format specifier that required a value.
`Variables defined as DLL imports cannot be defined with an initial value.`	Compile Error	A variable defined as a DLL import is being assigned an initial value, e.g. `int __import i = 0;` The variable must be initialized in a separate assignment statement.
`VXI address must be a multiple of 2 for word transfer.`	Fatal Runtime Error	An attempt was made to perform VXI word transfer beginning at an odd address.
`VXI address must be a multiple of 4 for longword transfer.`	Fatal Runtime Error	An attempt was made 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 Runtime Error	The w modifier cannot be used with the l format specifier.

(continues)

Table A-1. Error Messages (Continued)

Error Message	**Type Error**	**Comment**
`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	The specified module name is not present in the project or instrument list.
`z modifier only valid if rep is present.`	Non-Fatal Runtime Error	The `z` modifier cannot be used if the format specifier is not an array.
`z modifier required to match string parameter.`	Non-Fatal Runtime Error	If a character string must be treated as an array of another type, the `z` modifier must be used. This error may also be caused if the order of the arguments is incorrect, or if an argument is missing.

Appendix B
Error Checking in LabWindows/CVI

This appendix describes error checking codes in the LabWindows/CVI environment and how errors are reported in LabWindows/CVI libraries and instruments.

When you develop applications in LabWindows/CVI, you typically have debugging and **Break on Library Errors** enabled. With these utilities enabled, LabWindows/CVI identifies and reports programming errors in your source code, so you may have a tendency to be relaxed in your own error checking. However, in compiled modules and standalone executables, debugging and **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 due to 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 would return an error if `pnl` or `ctrl` were 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 are developing your program, and this function would not be affected by external factors. Therefore, it is generally not necessary to perform explicit error checking on this type of function.

One method of error checking is to check the status of function calls upon their completion. Many functions available from commercial libraries are designed to report any problems encountered while attempting to respond to user commands. The libraries available from LabWindows/CVI are no exception. All the functions in both the LabWindows/CVI libraries and 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 expected. In this appendix, you will see how these status codes are reported 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 a 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 that you can call to translate the integer value to an error string. When an error code is specific to a function, you can find its corresponding error string with the function description in the LabWindows/CVI manual set. The error string 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, both of these methods are used. In either case, the programmer should monitor these values for any report of an error and take appropriate action. A common technique for error checking is to monitor the status of functions, and if an error is reported, pause the program and report the error to the user by way of a pop-up message. For example, when a user interface panel is loaded into memory, a positive integer value is returned when the panel is successfully loaded. However, if a problem occurred, then the return value is negative. The following example shows an error message handler.

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

In a case when the status is reported by way of a global variable, such as in the RS-232 Library, the error checking is similar. In this case the global variable has a negative value when the function has failed.

```
bytesRead = ComRd (1, buffer, 10);
if (rs232err < 0) {
      ErrorCheck ("Error Reading From ComPort #1", rs232err,
                    GetRS232ErrorString(rs232err));
}
```

Notice that the above function also returned the number of bytes read from the serial port. The programmer could compare the number of bytes read to the number requested, and see if a discrepancy exists, then take the appropriate action. Notice also the way the error codes differ between the RS-232 Library and the User Interface Library. This difference is why it is important to always check the style and values of errors reported by the libraries used. A section on status reporting by library follows this section.

Once an error is detected, the program must take some action to either correct the situation or to 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

In this section, you will see how errors are reported in LabWindows/CVI libraries and instrument drivers. Notice, however, that those libraries which return their status code by global variables can also report more status information through return values.

User Interface Library

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

Analysis/Advanced Analysis Libraries

Both the Analysis and Advanced Analysis libraries return status information through a return value. If this return value is negative after the function returns, an error occurred. This library uses a common set of error codes, which are listed in the *LabWindows/CVI Standard Libraries Reference Manual*, the *LabWindows/CVI Advanced Analysis Reference Manual*, and the function panel help. You can use the function `GetAnalysisErrorString` to get the error message associated with the Analysis Library error code.

Data Acquisition Library

The NI-DAQ libraries have a large set of possible return codes indicating both errors and warnings. If the return code is negative, an error occurred. If the return code is positive, a warning is issued.

Note: ***The main difference between an error and a warning is that when the function detects an error it returns immediately. However, when a warning is detected, the function notes that fact, and continues execution. The warning simply informs the user that the function may not have executed as expected.***

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.

VXI Library

The VXI Library uses a variety of global variables and function return codes to report any error that occurs. You should check each function description to determine what error checking may be needed. 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: ***The GPIB Library returns*** `ibsta` ***as the return codes. However, because this value is also available as a global variable, the programmer must decide which one to check.***

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. Typically 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 global error codes and information on the individual functions. You can use the function `GetRS232ErrorString` to get the error message associated with a RS-232 Library error code.

TCP Library

The TCP library returns status information through a return value. If this return value is negative after the function returns, an error occurred. This library uses a common set of error codes, which are listed in the *LabWindows/CVI Standard Libraries Reference Manual* and the LabWindows/CVI function panel help. You can use the function `GetTCPErrorString` to get the error message associated with a TCP Library error code.

DDE Library

The DDE library returns status information through a return value. If this return value is negative after the function returns, an error occurred. This library uses a common set of error codes, which are listed in the *LabWindows/CVI Standard Libraries Reference Manual* and the LabWindows/CVI function panel help. You can use the function `GetDDEErrorString` to get the error message associated with a DDE Library error code.

X Property Library

The X Property library returns status information through a return value. If this return value is negative after the function returns, an error occurred. This library uses a common set of error codes, which are listed in the *LabWindows/CVI Standard Libraries Reference Manual* and the LabWindows/CVI function panel help. You can use the function `GetXPropErrorString` to get the error message associated with an X Property Library error code.

Formatting and I/O Library

This library contains the file I/O, string manipulation, and data formatting functions. All three sections can return negative error codes through their return values. However, an important fact must be kept in mind for the string functions. When debugging is enabled, the LabWindows/CVI environment keeps track of the sizes of strings and array. If any attempt to access a string or array beyond its boundary is detected, the environment halts the program and informs the user. It is important to remember that once you are in a compiled module or standalone executable, this feature no longer exists. The string functions can write beyond the end of a string or array without detection, resulting in corruption of memory. Therefore, you should 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 peformed 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

Error codes for the Utility functions are reported 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 associated with each return value.

ANSI C Library

Some of the ANSI C library functions report error codes as return values. Some functions also set the global variable `errno`. 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*. Alternatively, you can use the LabWindows/CVI function panel help to determine the error conditions assoicated with each return value.

LabWindows/CVI Instrument Drivers

Instrument drivers from National Instruments use a standard status reporting scheme. Error codes are reported as return values, and you can check each function individually in LabWindows/CVI function panel help to determine the error conditions associated with each return value.

Appendix C
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Electronic Services

Bulletin Board Support

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

- United States: (512) 794-5422 or (800) 327-3077
  Up to 14,400 baud, 8 data bits, 1 stop bit, no parity
- United Kingdom: 01635 551422
  Up to 9,600 baud, 8 data bits, 1 stop bit, no parity
- France: 1 48 65 15 59
  Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FaxBack Support

FaxBack is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access FaxBack from a touch-tone telephone at the following number: (512) 418-1111.

FTP Support

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

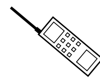
E-Mail Support (currently U.S. only)

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

GPIB:	`gpib.support@natinst.com`
DAQ:	`daq.support@natinst.com`
VXI:	`vxi.support@natinst.com`
LabVIEW:	`lv.support@natinst.com`
LabWindows:	`lw.support@natinst.com`
HiQ:	`hiq.support@natinst.com`
Lookout:	`lookout.support@natinst.com`
VISA:	`visa.support@natinst.com`

Fax and Telephone Support

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

	 Telephone	Fax
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

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

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

Name ____

Company ____

Address ____

____

Fax (_____)___________________________ Phone (_____)______________________________

Computer brand ______________ Model ________________ Processor ___________________

Operating system: Windows 3.1, Windows for Workgroups 3.11, Windows NT 3.1, Windows NT 3.5, Windows 95, other (include version number)____

Clock Speed __________MHz RAM ___________MB Display adapter _________________

Mouse ____yes ____no Other adapters installed ____

Hard disk capacity ______MB Brand ____

Instruments used ____

National Instruments hardware product model __________________ Revision __________________

Configuration ____

National Instruments software product ________________________ Version ___________________

Configuration ____

The problem is ____

____

____

____

List any error messages ____

____

____

____

The following steps will reproduce the problem ____

____

____

____

Hardware and Software Configuration Form

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

National Instruments Products

Data Acquisition Hardware Revision ______________________________

Interrupt Level of Hardware ______________________________

DMA Channels of Hardware ______________________________

Base I/O Address of Hardware ______________________________

NI-DAQ, LabVIEW, or
LabWindows Version ______________________________

Other Products

Computer Make and Model ______________________________

Microprocessor ______________________________

Clock Frequency ______________________________

Type of Video Board Installed ______________________________

Operating System ______________________________

Operating System Version ______________________________

Operating System Mode ______________________________

Programming Language ______________________________

Programming Language Version ______________________________

Other Boards in System ______________________________

Base I/O Address of Other Boards ______________________________

DMA Channels of Other Boards ______________________________

Interrupt Level of Other Boards ______________________________

Documentation Comment Form

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

Title: **LabWindows®/CVI Programmer Reference Manual**

Edition Date: **July 1996**

Part Number: **320685C-01**

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

____
____
____
____
____
____
____
____

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

____
____
____
____
____
____
____
____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

____

Fax (_____)___________________________ Phone (_____)_______________________________

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

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Glossary

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

A

API	Application Programming Interface.
active window	The window affected by user input at a given moment. The title of an active window is highlighted.
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 arguments are passed right to left.
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 <ENTER> is pressed and no command is specifically selected. Default command buttons are indicated in dialog boxes with a double outline.
dialog box	A prompt mechanism in which you specify additional information needed to complete a command.
DLL	Dynamic Link Library. A file containing a collection of functions that can be used by multiple applications (`.exe` files).

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 is ignored during compilation and execution. Excluded lines of code are displayed in a different color than included lines of code.

F

.fp file	A file containing information about the function tree and function panels of an instrument module.
full-screen mode	A screen display mode in which one window occupies the entire screen.
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. It is described in the *LabWindows/CVI Instrument Library Developer's 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 — The window in which you build the skeleton of a function panel file. It is described in the *LabWindows/CVI Instrument Library Developer's Guide*.

G

Generated Code box — A small box located at the bottom of the function panel screen that displays the code produced by the manipulation of 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 are not specifically returned 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 command that has no menu items associated with it and takes effect immediately when selected. An immediate action command is suffixed with an exclamation point (!).

input control — A function panel control that accepts a value typed 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.

Instrument Library — A LabWindows/CVI library that contains instrument control functions.

Interactive Execution window — A LabWindows/CVI work area in which sections of code may be executed 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 determined by the function you execute. 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. It is arbitrarily defined by the linker that creates the DLL or it may be specified in the `.def` file when the DLL is created.

P

PASCAL	A function calling convention in which function arguments are passed left to right.
primary control	A function panel control that specifies parameters in the function panel's primary function.
primary function	The function that performs the main task associated with a function panel. A function panel has only one primary function, but can have many secondary functions.
Project window	A windows containing a list of files used by your application.
prompt command	A command that requires additional information before it can be executed; a prompt command appears on a pull-down menu suffixed with three ellipses (…).

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	Software Development Kit.
scroll bars	Areas along the bottom and right sides of a window that show your relative position in the file. Scroll bars can be used with a mouse 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 will affect by moving the input focus (highlight) to a particular item or area.
shortcut key commands	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 which determines the value placed in the function call.
Source window	A LabWindows/CVI work area where programs are edited and executed.
split-screen mode	A screen display mode in which two windows occupy the screen at once; each window occupies one-half of the display.
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.
String Display window	A window for viewing and editing string variables and arrays.

T

text box	A dialog box item in which text is entered from the keyboard to complete a command.

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. It is described in the *LabWindows/CVI User Interface Reference Manual*.

V

Variable Display window	A window that shows the values of the variables that are currently defined.

W

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

Index

Numbers/Symbols

_cdecl calling convention qualifier, 1-6
__cdecl calling convention qualifier, 1-6
CVI macro, 1-4
_CVI_DEBUG_ macro, 1-4
_CVI_DLL_ macro, 1-5
_CVI_EXE_ macro, 1-4
_CVI_LIB_ macro, 1-5
__declspec(dllexport) calling convention qualifier, 1-6, 3-18
__declspec(dllimport) calling convention qualifier, 1-6
__DEFALIGN_ macro, 1-5
_export calling convention qualifier, 1-7
__export calling convention qualifier, 1-7
__FLAT__ macro, 1-5
_import calling convention qualifier, 1-7
__import calling convention qualifier, 1-7
_M_IX86_ macro, 1-5
_NI_BC_ macro, 1-5
_NI_i386_ macro, 1-4
_NI_mswin_ macro, 1-4, 6-1
_NI_mswin16_ macro, 1-4, 6-1
_NI_mswin32_ macro, 1-4, 1-5, 6-1
_NI_SC_ macro, 1-5
_NI_sparc_ macro, 1-4, 6-1
_NI_unix_ macro, 1-4, 6-1
_NI_VC_ macro, 1-5
_NI_WC_ macro, 1-5
__NT__ macro, 1-5
_stcall calling convention qualifier, 1-6, 3-17
__stcall calling convention qualifier, 1-6, 3-17 to 3-18, 3-20
_WIN32 macro, 1-5
__WIN32__ macro, 1-5
_WINDOWS macro, 1-5
16-bit source code, converting to 32-bit source code, 1-9 to 1-10
16-bit Windows DLLs. *See* Microsoft Windows 16-bit DLLs.
32-bit Borland or Symantec compiled modules under Windows, 4-2 to 4-3
32-bit source code
  converting 16-bit source code to 32-bit source code, 1-9 to 1-10
  DLL calling directly back into 32-bit code, 4-12 to 4-14
32-bit Watcom compiled modules under Windows 3.1, 4-1 to 4-2
32-bit Windows DLLs. *See* Microsoft Windows 32-bit DLLs.

A

.a files, using with standalone executables, 7-8
Add Files To DLL button, 7-12
Add Files To Executable button, 7-12
Advanced Analysis Library, B-3
Analysis Library, B-3
ANSI C Library
  include files, for Windows 95/NT, 3-9
  status reporting by, B-6
ANSI C specifications
  multiplatform application portability, 6-2
  non-ANSI LabWindows/CVI compiler keywords, 1-5
  using low-level I/O functions, 1-8
array indexing errors. *See* pointer protection errors.
array passing in glue code, 4-10
asynchronous callbacks, compiled modules using, 2-5
asynchronous DLL functions, 4-11 to 4-12
Auto-Load List command, Edit menu, 8-2

B

bit fields, Windows 32-bit DLLs, 3-5
Borland C/C++
  Borland or Symantec 32-bit compiled modules under Windows, 4-2 to 4-3
  creating 16-bit Windows DLLs, 4-21 to 4-22
  creating object and library files for use in LabWindows/CVI, 3-14 to 3-15
Break on Library Errors option, 1-15 to 1-16, 7-16, B-1
buffer retention by DLL glue code, 4-11 to 4-12
Build Error window, 1-4
Build menu
  Compiler Defines command, 1-4
  Create Distribution Kit command, 7-1, 7-8
  External Compiler Support command, 3-10
  Target command, 3-16, 3-17, 3-21
building platform-independent applications. *See* multiplatform applications, building.
bulletin board support, C-1

C

.c files. *See* source files.
C language extensions
  calling conventions (Windows 95/NT), 1-6 to 1-7
    import and export qualifiers, 1-6 to 1-7
  C++-style comment markers, 1-7
  duplicate typedefs, 1-7
  non-ANSI C standard keywords, 1-5
  program entry points (Windows), 1-8
  structure packing pragma (Windows), 1-7 to 1-8
C library
  using low-level I/O functions, 1-8
C++ style comment markers, 1-7
callback functions
  compiled modules using asynchronous callbacks, 2-5
  direct callback by DLLs, 4-12 to 4-14
  status change notification for compiled modules, 2-4 to 2-5
callback references, resolving (Windows 95/NT)
  from modules loaded at run-time, 3-11 to 3-12
    references to non-LabWindows/CVI symbols, 3-11
    run-time module references to symbols not exported from DLL, 3-12
  from .uir files, 3-9 to 3-10
    linking to callback functions not exported from DLL, 3-10
calling conventions (Windows 95/NT), 1-6 to 1-7
  for exported functions, 3-17 to 3-18
  import and export qualifiers, 1-6 to 1-7
casting. *See* pointer casting.
cdecl calling convention, 1-6
_cdecl calling convention qualifier, 1-6
__cdecl calling convention qualifier, 1-6
Check Disk Dates Before Each Run option, 4-4
CloseCVIRTE function, 3-13 to 3-14
code. *See* source files.
colors, multiplatform application considerations, 6-3
comment markers, C++ style, 1-7
Compatibility with option, 1-2
compiled modules. *See* loadable compiled modules.
compiler. *See also* compiler options.
  C library issues
    using low-level I/O functions, 1-8
  compiler defines, 1-4 to 1-5
  data types
    allowable data types (table), 1-9
    converting 16-bit code to 32-bit code, 1-9 to 1-10
  debugging levels, 1-10 to 1-11
  error messages, A-1 to A-58

limits (table), 1-1
non-ANSI C keywords, 1-5
overview, 1-1
user protection errors
general protection errors, 1-15
library protection errors, 1-15 to 1-16
memory corruption (fatal), 1-15
memory deallocation (non-fatal), 1-14
pointer arithmetic (non-fatal), 1-12
pointer assignment (non-fatal), 1-12
pointer casting (non-fatal), 1-14
pointer comparison (non-fatal), 1-13
pointer dereference errors (fatal), 1-13
pointer subtraction (non-fatal), 1-14
compiler defines
predefined macros
for platform-dependent code, 1-4
for Windows 95 and NT, 1-4 to 1-5
syntax, 1-4
Compiler Defines command
Build menu, 1-4
Options menu, 3-22
compiler options
Compatibility with, 1-2
compiled object modules
Borland C 4.x, 4-3
Symantec C++ 6.0, 4-2 to 4-3
Watcom, 4-2
Default calling convention option, 1-2
Display status dialog during build, 1-4
Enable signed/unsigned pointer mismatch warning, 1-3
Enable unreachable code warning, 1-3
Maximum number of compile errors, 1-2
Prompt for include file paths, 1-3 to 1-4
Require function prototypes, 1-2, 2-2, 2-3
Require return values for non-void functions, 1-2 to 1-3
Show Build Error window for warnings, 1-4
Stop on first file with errors, 1-4
Track include file dependencies, 1-3
Compiler Options command, Options menu, 3-4
Compiler Preferences command, Options menu, 1-2, 2-2, 2-3
compiler/linker issues. *See* specific operating system, e.g., UNIX operating system.
configuring Run-time Engine
cvidir option, 7-6 to 7-7
cvirt*x* option, 7-6
translating message file, 7-5
converting 16-bit source code to 32-bit source code, 1-9 to 1-10
Create Distribution Kit command, Build menu, 7-1, 7-8
Create Dynamic Link Library command, 3-20, 7-13 to 7-15
Create Object File command, Options menu, 3-21
Create Standalone Executable File command, 3-16, 7-13 to 7-16
Create Static Library command, 3-17, 3-21
creating
loadable compiled modules. *See* loadable compiled modules.
platform-independent applications. *See* multiplatform applications, building.
standalone executables. *See* standalone executables, creating and distributing.
Windows DLLs. *See* Microsoft Windows 16-bit DLLs; Microsoft Windows 32-bit DLLs.
customer communication, *xi,* C-1 to C-2
CVI macro, 1-4
_CVI_DEBUG macro, 1-4
cvidir configuration option (Windows 95/NT), 7-6 to 7-7
_CVI_DLL_ macro, 1-5
_CVI_EXE_ macro, 1-4
_CVI_LIB_ macro, 1-5
cvirt*x* configuration option (Windows 3.1), 7-6

D

Data Acquisition Library, B-3
data types
  allowable data types for compiler (table), 1-9
  converting 16-bit source code to 32-bit source code, 1-9 to 1-10
DDE Library, B-5
debugging levels
  Extended, 1-11
  None, 1-11
  Standard, 1-11
__declspec(dllexport) calling convention qualifier, 1-6, 3-18
__declspec(dllimport) calling convention qualifier, 1-6
__DEFALIGN_ macro, 1-5
Default calling convention option, 1-2
Display status dialog during build option, 1-4
distributing libraries, 8-1 to 8-3
  adding to user's Library menu, 8-1 to 8-2
  specifying library dependencies, 8-2 to 8-3
distributing standalone executables. *See* standalone executables, creating and distributing.
DLLEXPORT macro, 1-7, 3-18
DLLIMPORT macro, 1-7
DLLs. *See* Microsoft Windows 16-bit DLLs; Microsoft Windows 32-bit DLLs.
DLLSTDCALL macro, 3-18, 3-20
documentation
  conventions used in manual, *x*
  organization of manual, *ix-x*
  related documentation, *xi*
doubles, returning, 3-5
duplicate typedefs, 1-7
dynamic memory protection, 1-19
dynamic memory protection errors
  memory corruption (fatal), 1-15
  memory deallocation (non-fatal), 1-14

E

Edit menu, Function Tree Editor, 8-2
electronic support services, C-1 to C-2
e-mail support, C-2
Enable signed/unsigned pointer mismatch warning option, 1-3
Enable unreachable code warning option, 1-3
enum sizes, Windows 32-bit DLLs, 3-5
error checking, B-1 to B-6
  Break on Library Errors option, 1-15 to 1-16, 7-16, B-1
  enabling Require Function Prototypes option (caution), 1-2
  overview, B-1
  standalone executables, 7-16
  status codes
    checking function call status codes, B-2
    returned by LabWindows/CVI functions, B-2
  status reporting by libraries and instrument drivers, B-3 to B-6
errors. *See also* user protection errors.
  compiler-related error messages, A-1 to A-58
  enabling runtime error checking (caution), 1-2
  Maximum number of compile errors option, 1-2
  missing prototype errors, 1-2
  terminating compilation for file errors, 1-4
events, multiplatform application considerations, 6-3
executable file, required for standalone executables, 7-7
executables, creating and distributing. *See* standalone executables, creating and distributing.
_export calling convention qualifier, 1-7
__export calling convention qualifier, 1-7

export qualifiers
calling conventions
(Windows 95/NT), 1-6 to 1-7
exporting DLL functions and variables, 3-18 to 3-19
External Compiler Support command, Build menu, 3-10
external modules. *See also* loadable compiled modules.
definition, 2-3
forcing referenced modules into executable or DLL, 7-12
multiplatform application considerations, 6-3
under UNIX
compiling with external compilers, 5-3 to 5-4
restrictions, 5-3
using loadable compiled module as, 2-3

F

fax and telephone support, C-2
FaxBack support, C-1
files for running standalone executables
accessing UIR, image, and panel state files, 7-9
DLL files, 7-8
loading files using LoadExternalModule, 7-11 to 7-16
DLL files and DLL path files (Windows 3.1), 7-14 to 7-15
DLL files (Windows 95/NT), 7-14
files in project, 7-12 to 7-13
forcing referenced modules into executable or DLL, 7-12
library files not in project, 7-13
object files not in project, 7-13
other types of files, 7-16
source files, 7-15 to 7-16
location of files on target machine, 7-8 to 7-16
relative pathnames for accessing files, 7-16
required files, 7-7 to 7-8
__FLAT__ macro, 1-5
floats, returning, 3-5
fonts, multiplatform application considerations, 6-3
Formatting and I/O Library, B-5 to B-6
<FORWARD DELETE> key, multiplatform application considerations, 6-3
FTP support, C-2
full prototype, 1-2
function definition
new style, 1-2
old style, 1-2
function prototypes
occurrence of missing prototype errors, 1-2
requiring, 1-2
functions exported by ordinal value only, 4-18

G

general protection errors, 1-15
Generate DLL Glue Object command, Options menu, 7-14
Generate DLL Glue Source command, Options menu, 4-8, 4-9
Generate DLL Import Library command, Options menu, 3-4
Generate DLL Import Source command, Options menu, 3-16
Generate Windows Help command, Options menu, 3-20
GetCVIWindowHandle function, 4-20
glue code. *See* Microsoft Windows 16-bit DLLs.
GPIB/GPIB 488.2 Library, B-4
graphical user interface (GUI), multiplatform application considerations, 6-3

H

hardware interrupts under Windows 95/NT, 3-24 to 3-25
hot keys. *See* shortcut keys.

I

image files
- accessing from standalone executables, 7-9
- multiplatform application considerations, 6-3
- using with standalone executables, 7-8

_import calling convention qualifier, 1-7
__import calling convention qualifier, 1-7
import libraries (Windows 95/NT)
- automatic loading of SDK import libraries, 3-23
- compatibility with external compilers, 3-4
- customizing DLL import libraries, 3-16 to 3-17
- generating DLL import library, 3-3
- link errors when using DLL import libraries, 3-2

import qualifiers. *See also* export qualifiers.
- calling conventions (Windows 95/NT), 1-6 to 1-7
- marking imported symbols in include file, 3-19

include file dependencies
- prompting for paths, 1-3 to 1-4
- tracking, 1-3

include files
- ANSI C library and LabWindows/CVI libraries, 3-9
- generating glue code, 4-9
- Windows 32-bit DLLs
  - exporting DLL functions and variables, 3-18
  - marking imported symbols in include file, 3-19
- Windows SDK functions, 3-22

include paths, setting up for LabWindows/CVI, ANSI C, and SDK libraries, 3-23 to 3-24
Include Paths command, Options menu, 1-20
InitCVIRTE, calling
- UNIX executables, 5-2 to 5-3
- Windows 95/NT executables, 3-13 to 3-14

Instrument Directories command, Options menu, 8-2
instrument drivers
- definition, 2-2
- status reporting, B-6
- using loadable compiled modules as program files, 2-2

Instrument menu, 2-2, 7-12
interrupts under Windows 95/NT, 3-24 to 3-25

K

keywords. *See also* **Numbers/Symbols.**
- non-ANSI LabWindows/CVI compiler keywords, 1-5
- void keyword, 1-2

L

LabWindows/CVI compiler. *See* compiler.
LabWindows/CVI Run-time Engine. *See* Run-time Engine.
.lib files. *See* library files.
libraries
- C library issues, 1-8
- creating static libraries, 3-21
- distributing, 8-1 to 8-3
  - adding to user's Library menu, 8-1 to 8-2
  - specifying library dependencies, 8-2 to 8-3
- loading library files for standalone executables, 7-13
- portability issues for multiplatform applications, 6-1 to 6-2

using loadable compiled modules as user libraries, 2-2 to 2-3
Windows 95/NT compiler issues
calling InitCVIRTE and CloseCVIRTE, 3-13 to 3-14
include files for ANSI C library and LabWindows/CVI libraries, 3-9
multithreading and LabWindows/CVI libraries, 3-7 to 3-8
resolving callback references from .uir files, 3-9 to 3-10
resolving references from modules loaded at run-time, 3-11 to 3-12
run state change callbacks unavailable, 3-12
standard input/output windows, 3-9
using LabWindows/CVI libraries in external compilers, 3-8 to 3-13
library files
compatibility with external compilers (Windows 95/NT), 3-4
creating in external compilers for use in LabWindows/CVI, 3-14 to 3-15
loading with LoadExternalModule, 7-13
using with standalone executables, 7-8
library function user protection errors, 1-19
disabling, 1-17 to 1-18
Library menu
appearance of user libraries on, 2-3
installing user libraries, 2-2, 8-1 to 8-2
linking modules with external modules, 7-12
Library Options command, Project Options menu, 2-2, 8-1
library protection errors, 1-15 to 1-16
disabling
at run-time, 1-16
for functions, 1-17 to 1-18
errors involving library protection, 1-15
#line preprocessor directive, 1-5
loadable compiled modules
16-bit Windows DLLs
creating
with Borland C++, 4-21 to 4-22
with Microsoft Visual C++ 1.5, 4-21
glue code
DLLs unable to use glue code generated at load time, 4-8 to 4-19
DLLs using glue code generated at load time, 4-8
requirements, 4-7 to 4-8
helpful LabWindows/CVI options, 4-4
overview, 4-4
rules and restrictions, 4-5 to 4-7
search precedence, 4-22 to 4-23
32-bit Borland or Symantec compiled modules under Windows, 4-2 to 4-3
32-bit Watcom compiled modules under Windows 3.1, 4-1 to 4-2
advantages and disadvantages, 2-1 to 2-2
external modules, 2-3
instrument driver program files, 2-2
modules compiled by LabWindows/CVI, 4-1
multiplatform application considerations, 6-3
overview, 2-1
project list, 2-3
requirements, 2-1
special considerations, 2-4 to 2-5
status changes
examples of program state changes, 2-4 to 2-5
modules using asynchronous callbacks, 2-5
notifying compiled modules of changes, 2-4 to 2-5
user libraries, 2-2 to 2-3
Windows messages passed from DLLs, 4-19 to 4-20
GetCVIWindowHandle function, 4-20
RegisterWinMsgCallback function, 4-19 to 4-20
UnRegisterWinMsgCallback function, 4-20
LoadExternal Module for loading files, 7-11 to 7-16
DLL files and DLL path files (Windows 3.1), 7-14 to 7-15

DLL files (Windows 95/NT), 3-2, 7-14
files listed in project, 7-12 to 7-13
forcing modules into executable or DLL, 7-12
library files not in project, 7-13
object files not in project, 7-13
other types of files, 7-16
source files, 7-15 to 7-16
locking process segments into memory using plock(), 5-4
long doubles, Windows 32-bit DLLs, 3-6
low-level I/O functions, using, 1-8

M

macros, predefined
Windows 95/NT, 1-4 to 1-5
writing platform-independent code, 1-4, 6-1
manual. *See* documentation.
math coprocessor software emulation for Windows 3.1, 7-2
Maximum number of compile errors option, 1-2
memory protection errors
memory corruption (fatal), 1-15
memory deallocation (non-fatal), 1-14
message file for Run-time Engine, translating, 7-5
messages passed from DLLs. *See* Microsoft Windows messages passed from DLLs.
Microsoft Visual Basic, automatic inclusion of Type Library resource for, 3-20 to 3-21
Microsoft Visual C++
creating 16-bit Windows DLLs, 4-21
creating object and library files for use in LabWindows/CVI, 3-14
Microsoft Windows 3.1
compiler/linker issues
16-bit Windows DLLs. *See* Microsoft Windows 16-bit DLLs.
32-bit Borland or Symantec compiled modules, 4-2 to 4-3
32-bit Watcom compiled modules, 4-1 to 4-2
modules compiled by LabWindows/CVI, 4-1
cvirt*x* option for configuring Run-time Engine, 7-6
distributing standalone executables
math coprocessor software emulation, 7-2
minimum system requirements, 7-2
program entry points, 1-8
structure packing pragmas, 1-7 to 1-8
Microsoft Windows 16-bit DLLs
creating
with Borland C++, 4-21 to 4-22
with Microsoft Visual C++ 1.5, 4-21
fixing linker error (note), 4-7
for standalone executables
definition, 7-8
loading with LoadExternalModule, 7-14 to 7-15
rules for using, 7-10 to 7-11
glue code
DLLs unable to use glue code generated at load time, 4-8 to 4-19
arrays bigger than 64 K, 4-10
buffer retained after function returns (asynchronous function), 4-11 to 4-12
direct callbacks into 32-bit code, 4-12 to 4-14
functions exported by ordinal value only, 4-18
loading, 4-8 to 4-9
pointer that points to other pointers, 4-16 to 4-18
returning pointers, 4-14 to 4-16
rules for include file, 4-9
support module required outside of DLL, 4-9
DLLs using glue code generated at load time, 4-8
requirements, 4-4
unusable in specific situations, 4-7 to 4-8
helpful LabWindows/CVI options, 4-4
not supported in Windows 95/NT, 3-2
overview, 4-4
rules and restrictions, 4-5 to 4-7
search precedence, 4-22 to 4-23

Microsoft Windows 32-bit DLLs
compatibility with external compilers, 3-4 to 3-6
bit fields, 3-5
enum sizes, 3-5
long doubles, 3-5
returning floats and doubles, 3-5
returning structures, 3-5
structure packing, 3-4 to 3-5
creating in LabWindows/CVI, 3-16 to 3-21
automatic inclusion of Type Library resource for Visual Basic, 3-20 to 3-21
calling conventions for exported functions, 3-17 to 3-18
customizing import library, 3-16 to 3-17
exporting DLL functions and variables, 3-18 to 3-19
export qualifier method, 3-18 to 3-19
include file method, 3-18
marking imported symbols in include file distributed with DLL, 3-19 to 3-20
preparing source code, 3-17 to 3-20
recommendations, 3-20
DLL import library compatibility with external compilers, 3-4
for standalone executables
distributing, 7-8
loading with LoadExternalModule, 7-14
location, 7-9
rules for using, 7-10
loading, 3-1 to 3-3
16-bit DLLs not supported, 3-2
default unloading/reloading policy, 3-3
DLL path (.pth) files not supported, 3-2
DLLs for instrument drivers and user libraries, 3-2
generating import library, 3-3
link errors when using DLL import libraries, 3-2
using LoadExternalModule function, 3-2
Microsoft Windows 95/NT
32-bit DLLS. *See* Microsoft Windows 32-bit DLLs.
calling convention qualifiers in function declarations, 1-6 to 1-7
calling Windows SDK functions in LabWindows/CVI, 3-22 to 3-23
automatic loading of SDK import libraries, 3-23
creating multiple threads using Windows SDK functions, 3-23
interface capabilities of Windows SDK functions, 3-22
Windows SDK include files, 3-22
compatibility with external compilers, 3-3 to 3-6
choosing a compiler, 3-4
Compatibility with option, 1-2
Default calling convention option, 1-2
DLLs, 3-4 to 3-6
external compiler versions supported, 3-6
LabWindows/CVI differences, 3-6
object files, library files, and DLL import libraries, 3-4
required preprocessor definitions, 3-6
creating executables in LabWindows/CVI, 3-16
creating object and library files in external compiler, 3-14 to 3-15
Borland C/C++ command line compiler, 3-14 to 3-15
Microsoft Visual C/C++, 3-14
Symantec C/C++, 3-15
Watcom C/C++, 3-15
creating object files in LabWindows/CVI, 3-21
creating static libraries in LabWindows/CVI, 3-21
cvidir option for configuring Run-time Engine, 7-6 to 7-7

distributing standalone executables
coprocessor not required, 7-2
location of files, 7-9
minimum system requirements, 7-1
hardware interrupts, 3-24 to 3-25
LabWindows/CVI libraries in external compilers, 3-8 to 3-14
calling InitCVIRTE and CloseCVIRTE, 3-13 to 3-14
include files, 3-9
resolving callback references from .uir files, 3-9 to 3-10
linking to callback functions not exported from DLL, 3-10
resolving references from modules loaded at run-time, 3-11 to 3-12
references to non-LabWindows/CVI symbols, 3-11
references to symbols not exported from DLL, 3-12
run state change callbacks unavailable, 3-12
standard input/output window, 3-9
multithreading and LabWindows/CVI libraries, 3-7
program entry points, 1-8
setting up include paths for LabWindows/CVI, ANSI C, and SDK libraries, 3-23 to 3-24
structure packing pragmas, 1-7 to 1-8
Microsoft Windows messages passed from DLLs, 4-19 to 4-20
GetCVIWindowHandle function, 4-20
RegisterWinMsgCallback function, 4-19 to 4-20
UnRegisterWinMsgCallback function, 4-20
Microsoft Windows SDK functions, 3-22 to 3-23
automatic loading of SDK import libraries, 3-23
calling in LabWindows/CVI, 3-22 to 3-23
creating multiple threads, 3-23
include files, 3-22
setting up include paths for SDK libraries, 3-23 to 3-24
user interface capabilities, 3-22
minimum system requirements for standalone executables, 7-1 to 7-2
missing return value (non-fatal) error, 1-15
_M_IX86_ macro, 1-5
modini program (caution), 8-2, 8-3
modreg program (caution), 8-2, 8-3
multiplatform applications, building
externally compiled module issues, 6-3
library portability issues, 6-1 to 6-2
predefined macros, 1-4 to 1-5, 6-1
programming guidelines, 6-1 to 6-3
user interface guidelines, 6-3
multithreading
creating multiple threads with Windows SDK functions, 3-23
using LabWindows/CVI libraries, 3-7 to 3-8

N

_NI_BC_ macro, 1-5
_NI_i386_ macro, 1-4
_NI_mswin_ macro, 1-4, 6-1
_NI_mswin16_ macro, 1-4, 6-1
_NI_mswin32_ macro, 1-4, 1-5, 6-1
_NI_SC_ macro, 1-5
_NI_sparc_ macro, 1-4, 6-1
_NI_unix_ macro, 1-4, 6-1
_NI_VC_ macro, 1-5
_NI_WC_ macro, 1-5
non-void functions, requiring return values for, 1-2
__NT__ macro, 1-5

O

.o files
  loading with LoadExternalModule, 7-11
  using with standalone executables, 7-8
object files
  compatibility with external compilers (Windows 95/NT), 3-4
  creating
    in external compilers for use in LabWindows/CVI, 3-14 to 3-15
    in LabWindows/CVI, 3-21
  loading with LoadExternalModule, 7-11
  using with standalone executables, 7-8
Options menu
  Function Tree Editor
    Generate Windows Help command, 3-20
  Project window
    Compiler Defines command, 3-22
    Compiler Options command, 3-4
    Compiler Preferences command, 1-2, 2-2, 2-3
    Include Paths command, 1-20
    Instrument Directories command, 8-2
    Run Options command, 1-10, 1-15, 1-20
  Source, Interactive Execution, and Standard Input/Output windows
    Create Object File command, 3-21
    Generate DLL Glue Object command, 7-14
    Generate DLL Glue Source command, 4-8, 4-9
    Generate DLL Import Library command, 3-4
    Generate DLL Import Source command, 3-16
ordinal value for exporting functions, 4-18

P

pack pragma (Windows), 1-7 to 1-8, 3-4 to 3-5
panel state files
  accessing from standalone executables, 7-9
  required for standalone executables, 7-8
pascal, Pascal, and _pascal keywords, 1-5
Pascal DLL functions, 4-8, 4-9
PCX files, multiplatform application considerations, 6-3
platform-independent applications, building. *See* multiplatform applications, building.
plock function, UNIX, 5-4
pointer casting, 1-18 to 1-19
pointer mismatch warning, enabling, 1-3
pointer protection errors, 1-11 to 1-14
  disabling for individual pointers, 1-16 to 1-17
  dynamic memory protection errors, 1-14 to 1-15
  pointer arithmetic (non-fatal), 1-12
  pointer assignment (non-fatal), 1-12
  pointer casting (non-fatal), 1-14
  pointer comparison (non-fatal), 1-13
  pointer dereference errors (fatal), 1-13
  pointer subtraction (non-fatal), 1-14
pointers
  DLLs passing pointers that point to other pointers, 4-16 to 4-18
  returned by DLLs, 4-14 to 4-16
pragmas
  disabling or enabling library protection errors, 1-17 to 1-18
  structure packing (Windows), 1-7 to 1-8, 3-4 to 3-5
predefined macros
  Windows 95/NT, 1-4 to 1-5
  writing platform-independent code, 1-4 to 1-5, 6-1
printf function, using with external compiler, 3-9
process segments, locking into memory using plock(), 5-4
program entry points (Windows), 1-8

Project window, Run Options menu, 4-4
projects. *See also* source files.
- calling compiled modules in project list, 2-3
- loading project files with LoadExternalModule, 7-12 to 7-13

Prompt for include file paths option, 1-3 to 1-4
.pth files
- loading with LoadExternalModule, 7-11
- not supported for Windows 95/NT, 3-2
- using with standalone executables, 7-8, 7-10 to 7-11

Q

Q387 coprocessor emulation software (Quickware), 7-2

R

references, resolving. *See* callback references, resolving.
RegisterWinMsgCallback function, 4-19 to 4-20
Reload DLLs Before Each Run option, 3-3, 4-4
Require function prototypes option, 1-2, 2-2, 2-3
Require return values for non-void functions option, 1-2 to 1-3
resolving references. *See* callback references, resolving.
return values
- missing return value (non-fatal) error, 1-15
- requiring for non-void functions, 1-2 to 1-3

RS-232 Library, B-4
Run Options command, Options menu
- Break on library errors option, 1-15 to 1-16, 7-16
- Reload DLLs Before Each Run option, 3-3
- setting debugging levels, 1-10
- setting maximum stack size, 1-20
- Unload DLLs After Each Run option, 3-3

Run Options menu, Project window, 4-4
run state change notification for compiled modules
- asynchronous callbacks, 2-5
- examples of program state changes, 2-4 to 2-5
- prototype for callback, 2-4
- requirements, 2-4
- unavailable
  - for executables under UNIX, 5-2
  - for external compilers under Windows 95/NT, 3-12

Run-time Engine. *See also* standalone executables, creating and distributing.
- configuring, 7-5 to 7-7
  - cvidir option, 7-6 to 7-7
  - cvirt*x* option, 7-6
  - translating message file, 7-5
- files required for running executable programs, 7-7 to 7-8
- location and type of files
  - for Windows 3.1, 7-9
  - for Windows 95/NT, 7-9
- overview, 7-1
- system requirements, 7-1 to 7-2

runtime error checking, enabling (caution), 1-2

S

scanf function, using with external compiler, 3-9
SDK functions. *See* Microsoft Windows SDK functions.
search precedence of Windows DLLs, 4-22 to 4-23
shortcut keys, multiplatform application considerations, 6-3
Show Build Error window for warnings option, 1-4

Solaris distribution of standalone executables
  Solaris 1, 7-4 to 7-5
  Solaris 2, 7-3 to 7-4
source files
  converting 16-bit source code to 32-bit source code, 1-9 to 1-10
  enabling unreachable code warning, 1-3
  loading with LoadExternalModule, 7-11
  preparing for use in Windows 32-bit DLL, 3-17 to 3-20
    calling conventions for exported functions, 3-17 to 3-18
    exporting DLL functions and variables, 3-18 to 3-19
      export qualifier method, 3-18 to 3-19
      include file method, 3-18
    marking imported symbols in include file distributed with DLL, 3-19 to 3-20
    recommendations, 3-20
stack overflow error (fatal), 1-15
stack size, 1-20
standalone executables, creating and distributing
  accessing UIR, image, and panel state files, 7-9
  configuring Run-time Engine, 7-5 to 7-7
    option descriptions, 7-6 to 7-7
    translating message file, 7-5
  distributing
    under Solaris 1, 7-4 to 7-5
    under Solaris 2, 7-3 to 7-4
    under UNIX, 7-2 to 7-5
    under Windows 3.1, 7-2
    under Windows 95/NT, 7-1 to 7-2
  DLL files, 7-10 to 7-11
  error checking, 7-16
  loading files using LoadExternal Module, 7-11 to 7-16
    DLL files and DLL path files (Windows 3.1), 7-14 to 7-15
    DLL files for Windows 95/NT, 7-14
    library files, 7-13
    object modules, 7-13
    source files, 7-15 to 7-16
  location of files on target machine, 7-8 to 7-16
    DLL files
      Windows 3.1, 7-10 to 7-11
      Windows 95/NT, 7-10
    LabWindows/CVI Run-time Engine for Windows, 7-9
    loading files using LoadExternalModule, 7-11 to 7-16
    UIR, image, and panel state files, 7-9
  math coprocessor software emulation for Windows 3.1, 7-2
  relative pathnames for accessing files, 7-16
  system requirements, 7-1 to 7-2
  UNIX compiler/linker issues, 5-1 to 5-3
    InitCVIRTE called by main function, 5-2 to 5-3
    run state change callbacks not available, 5-2
  Windows 95/NT, 3-16
    necessary files, 7-8
standard input/output windows, LabWindows/CVI, 3-9
state change notification for compiled modules. *See* run state change notification for compiled modules.
state files. *See* panel state files.
static libraries, creating, 3-21
status codes
  checking function call status codes, B-2
  definition, B-1
  returned by LabWindows/CVI functions, B-2
status dialog, displaying, 1-4
status reporting by libraries and instrument drivers, B-3 to B-6
  Advanced Analysis Library, B-3
  Analysis Library, B-3
  ANSI C Library, B-6
  Data Acquisition Library, B-3
  DDE Library, B-5
  Formatting and I/O Library, B-5 to B-6

GPIB/GPIB 488.2 Library, B-4
LabWindows/CVI instrument drivers, B-6
RS-232 Library, B-4
TCP Library, B-5
User Interface Library, B-3
Utility Library, B-6
VXI Library, B-4
X Property Library, B-5
_stcall calling convention qualifier, 1-6, 3-17
__stcall calling convention qualifier, 1-6, 3-17 to 3-18, 3-20
stdcall calling convention, 1-6
Stop on first file with errors option, 1-4
structure packing pragmas (Windows), 1-7 to 1-8, 3-4 to 3-5
Sun C library. *See* UNIX C library.
support modules for glue code, 4-9
Symantec C/C++
creating object and library files for use in LabWindows/CVI, 3-15
Symantec or Borland 32-bit compiled modules under Windows, 4-2 to 4-3
system requirements for standalone executables, 7-1 to 7-2

T

Target command, Build menu, 3-16, 3-17, 3-21
TCP Library, B-5
technical support, C-1 to C-2
Track include file dependencies option, 1-3
Type Library resource for Visual Basic, 3-20 to 3-21
typedefs, duplicate, 1-7

U

.uir files. *See* user interface resource (.uir) files.
unions, 1-19
UNIX C library
calling Sun C library from source code, 5-1
restrictions, 5-1
using low-level I/O functions, 1-8
UNIX operating system
compiler/linker issues, 5-1 to 5-4
calling Sun C library functions, 5-1
InitCVIRTE called by main function, 5-2 to 5-3
restrictions, 5-1
creating executables, 5-1 to 5-3
run state change callbacks not available, 5-2
InitCVIRTE called by main function, 5-2 to 5-3
locking process segments in memory using plock(), 5-4
run state change callbacks not available, 5-2
using externally compiled modules, 5-3 to 5-4
compiling, 5-3 to 5-4
restrictions, 5-3
distribution of standalone executables, 7-2 to 7-5
minimum system requirements, 7-5
under Solaris 1, 7-4 to 7-5
under Solaris 2, 7-3 to 7-4
Unload command, Instruments menu, 2-2
Unload DLLs After Each Run option, 3-3
unreachable code warning, enabling, 1-3
UnRegisterWinMsgCallback function, 4-20
user interface. *See* graphical user interface (GUI).
user interface events. *See* events.
User Interface Library, B-3
user interface resource (.uir) files
accessing from running standalone executables, 7-9
multiplatform application considerations, 6-3

required for running standalone executables, 7-8
resolving callback references from, 3-9 to 3-10
linking to callback functions not exported from DLL, 3-10
user libraries. *See also* libraries.
installing, 2-2
similarity with instrument driver, 2-2
using loadable compiled modules, 2-2 to 2-3
user protection
dynamic memory, 1-19
library functions, 1-19
pointer casting, 1-18 to 1-19
stack size, 1-20
unions, 1-19
user protection errors
disabling, 1-16 to 1-18
at run-time, 1-16
for individual pointer, 1-16 to 1-17
library errors
at run-time, 1-16
for functions, 1-17 to 1-18
error category, 1-11
general protection errors, 1-15
library protection errors, 1-15 to 1-16
memory corruption (fatal), 1-15
memory deallocation (non-fatal), 1-14
pointer arithmetic (non-fatal), 1-12
pointer assignment (non-fatal), 1-12
pointer casting (non-fatal), 1-14
pointer comparison (non-fatal), 1-13
pointer dereference errors (fatal), 1-13
pointer subtraction (non-fatal), 1-14
severity level, 1-11
Utility Library, B-6

V

va_arg (ap, type), 1-5
Visual Basic, automatic inclusion of Type Library resource for, 3-20 to 3-21
Visual C++
creating 16-bit Windows DLLs, 4-21
creating object and library files for use in LabWindows/CVI, 3-14
void keyword, 1-2
VXI Library, B-4

W

Watcom C/C++
32-bit compiled modules under Windows 3.1, 4-1 to 4-2
creating object and library files for use in LabWindows/CVI, 3-15
Watcom WEMU387.386 coprocessor emulation software, 7-2
_WIN32 macro, 1-5
__WIN32__ macro, 1-5
Windows 95/NT
predefined macros, 1-4 to 1-5
Windows DLLS. *See* Microsoft Windows 16-bit DLLs; Microsoft Windows 32-bit DLLs.
_WINDOWS macro, 1-5

X

X Property Library, status reporting by, B-5