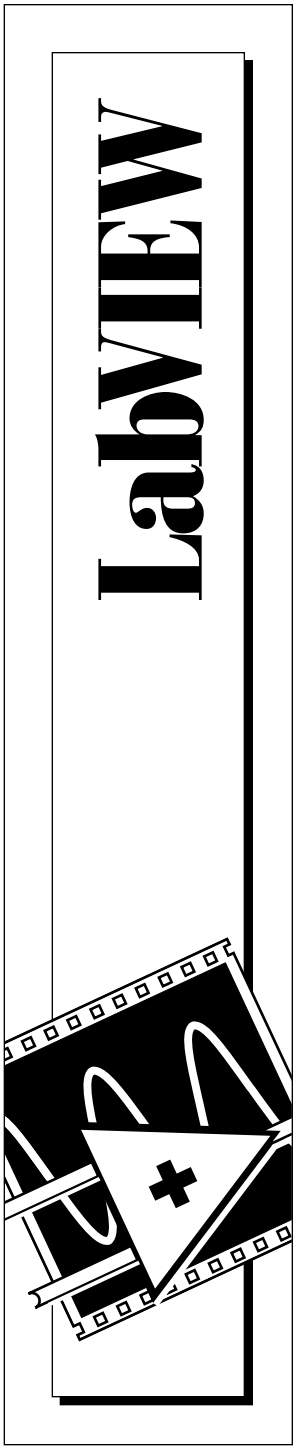


LabVIEW® Code Interface Reference Manual

November 1995 Edition
Part Number 320539C-01

Internet Support

GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabVIEW: lv.support@natinst.com
LabWindows: lw.support@natinst.com
HiQ: hiq.support@natinst.com

E-mail: info@natinst.com
FTP Site: ftp.natinst.com
Web Address: http://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 or (800) 329-7177

Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678 or (800) 328-2203

International Offices

Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 90 527 2321, France 1 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,
Italy 02 48301892, Japan 03 5472 2970, Korea 02 596 7456, Mexico 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

Important Information

Warranty

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

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

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

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

Copyright

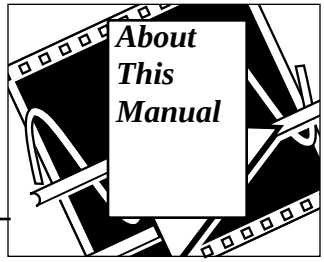
Trademarks

LabVIEW® is a trademark of National Instruments Corporation.

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

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

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



The *LabVIEW Code Interface Reference Manual* discusses Code Interface Nodes and external subroutines for users who need to access code written in conventional programming languages. The manual includes information about shared external subroutines, libraries of functions, memory and file manipulation routines, and diagnostic routines.

Organization of This Manual

This manual is organized as follows:

- Chapter 1, *CIN Overview*, introduces the LabVIEW Code Interface Node (CIN), a node that links external code written in a conventional programming language to LabVIEW.
- Chapter 2, *CIN Parameter Passing*, describes the data structures that LabVIEW uses when passing data to a CIN.
- Chapter 3, *CIN Advanced Topics*, covers several topics that are needed only in advanced applications, including how to use the `CINInit`, `CINDispose`, `CINAbort`, `CINLoad`, `CINUnload`, and `CINSave` routines. The chapter also discusses how global data works within CIN source code, and how users of Windows 3.1, Windows 95, and Windows NT can call a DLL from a CIN.
- Chapter 4, *External Subroutines*, describes how to create and call shared external subroutines from other external code modules.
- Chapter 5, *Manager Overview*, gives an overview of the function libraries, called *managers*, which you can use in external code modules. These include the memory manager, the file manager, and the support manager. The chapter also introduces many of the basic constants, data types, and globals contained in the LabVIEW libraries.
- Appendix A, *CIN Common Questions*, answers some of the questions commonly asked by LabVIEW CIN users.

- Appendix B, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including acronyms, abbreviations, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

bold	Bold text denotes menus, palettes, menu items, or dialog box buttons or options. In addition, bold text denotes VI input and output parameters.
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`	Monospace font denotes text or characters that you enter using the keyboard. Sections of code, programming examples, syntax examples, and messages and responses that the computer automatically prints to the screen also appear in this font.
`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 on the keyboard—for example, <`Shift`>.
-	A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys—for example, <`Shift-Delete`>.
paths	Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files, as in `drivename\dir1name\dir2name\myfile`.

Warning: ***This icon to the left of bold italicized text denotes a warning, which alerts you to the possibility of damage to you or your equipment.***

Caution: ***This icon to the left of bold italicized text denotes a caution, which alerts you to the possibility of data loss or a system crash.***

☞ **Note:** ***This icon to the left of bold italicized text denotes a note, which alerts you to important information.***

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

Related Documentation

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

- Your LabVIEW analysis VI reference manual
- *LabVIEW Instrument I/O VI Reference Manual*
- Your *LabVIEW Tutorial*
- Your *LabVIEW User Manual*

Sun users may also find the following document useful:

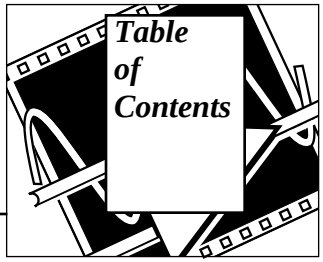
- *SPARCompiler C 3.0 Answer Book* CD-ROM, Sun Microsystems, Inc., U.S.A., 1993

Windows users may also find the following documents useful:

- Microsoft Windows documentation set, Microsoft Corporation, Redmond, WA, 1992-1995
- *Microsoft Windows Programmer's Reference*, Microsoft Corporation, Redmond, WA, 1992-1995
- *Win32 Programmer's Reference*, Microsoft Corporation, Redmond, WA, 1992-1995
- *Watcom C/C++ User's Guide* CD-ROM, Watcom Publications Limited, Waterloo, Ontario, Canada, 1995; Help file: "The Watcom C/C++ Compilers"

Customer Communication

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



About This Manual

Chapter 1
CIN Overview

Chapter 4
External Subroutines

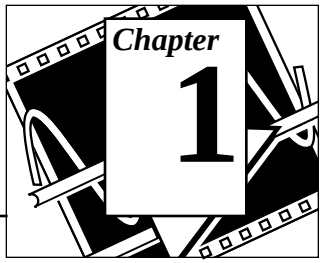
Chapter 5
Manager Overview

Chapter 6
Memory Manager Functions

Chapter 7
File Manager Functions

Figures

Tables



CIN Overview

This chapter introduces the LabVIEW Code Interface Node (CIN), a node that links external code written in a conventional programming language to LabVIEW.

Introduction

A CIN is a block diagram node associated with a section of source code written in a conventional programming language. You compile the source code first and link it to form executable code. LabVIEW calls the executable code when the node executes, passing input data from the block diagram to the executable code, and returning data from the executable code to the block diagram.

The LabVIEW compiler can usually generate code that is fast enough for most of your programming tasks. However, you can use CINs for tasks that a conventional language can accomplish more easily, such as tasks that are time-critical or require a great deal of data manipulation. CINs are also useful for tasks that you cannot perform directly from the diagram, such as calling system routines for which no corresponding LabVIEW functions exist. CINs can also link existing code to LabVIEW, although you may have to modify the code so that it uses the correct LabVIEW data types.

CINs execute synchronously. This means that while CIN code executes, no other LabVIEW processes can execute. Normally, when a VI executes, LabVIEW monitors menus and the keyboard and allows other applications to execute. LabVIEW also allows more than one VI to run simultaneously. However, when CIN object code executes, it takes control of the process, so that LabVIEW ignores keyboard events, menu clicks, and other diagrams. On the Macintosh and under Windows 3.1, CINs even prevent other applications from executing. Although you can create VIs that use CINs and behave in a more asynchronous fashion, be aware of this potential problem if you intend to write a CIN that will execute a long task and you need LabVIEW to multitask in the interim.

A CIN appears on the diagram as an icon with input and output terminals. You associate this node with a piece of code you want LabVIEW to call. When it is time for the node to execute, LabVIEW calls the code associated with the CIN, passing it the specified data.

In some cases, you may want a CIN to perform additional actions at certain execution times. For instance, you may want to initialize some data structures at load time or free private data structures when the user closes the VI containing the CIN. For these situations, you can write routines that LabVIEW calls at predefined times or when the node executes. Specifically, LabVIEW calls certain routines when the VI containing the CIN is loaded, saved, closed, aborted, or compiled. You generally use these routines in CINs that perform an on-going action, such as accumulating results from call to call, so that you can allocate, initialize, and deallocate resources at the correct time. Most CINs perform a specific action at run time only.

After you have written your first CIN as described in this manual, writing new CINs is relatively easy. The work involved in writing new CINs is mostly in coding the algorithm, because the interface to LabVIEW remains the same, no matter what the development system.

Classes of External Code

LabVIEW supports code resources for CINs and external subroutines.

An external subroutine is a section of code that you can call from other external code. If you write multiple CINs that call the same subroutine, you may want to make the shared subroutine an external subroutine. The code for an external subroutine is a separate file; when LabVIEW loads a section of external code that references an external subroutine, it also loads the appropriate external subroutine into memory. Using an external subroutine makes each section of calling code smaller, because the external subroutine does not require embedded code. Further, you need to make changes only once if you want to modify the subroutine.

☞ **Note:** ***LabVIEW does not support code resources for external subroutines on the Power Macintosh. If you are working with a Power Macintosh, you should use shared libraries instead of external subroutines. For information on building shared libraries, consult your development environment documentation.***

Although LabVIEW for Solaris 2.x and HP-UX support external routines, it is recommended that you use UNIX shared libraries instead, because they are a more standard library format.

Supported Languages

The interface for CINs and external subroutines supports a variety of compilers, although not all compilers can create code in the correct executable format.

External code must be compiled as a form of executable that is appropriate for a specific platform. The code must be relocatable, because LabVIEW loads external code into the same memory space as the main application.

Macintosh

LabVIEW for the Macintosh uses external code as a customized code resource (for 68K) or shared library (for Power Macintosh) that is prepared for LabVIEW using the separate utilities `lvsbutil.app` for THINK C and Metrowerks CodeWarrior, and `lvsbutil.tool` for the Macintosh Programmer's Workshop. These utilities are included with LabVIEW.

The LabVIEW utilities and object files are known to be compatible with the three major C development environments for the Power Macintosh, which are as follows:

- THINK C, versions 5, 6, and 7, and Symantec C++ version 8 for Power Macintosh, from Symantec Corporation of Cupertino, CA
- Metrowerks CodeWarrior from Metrowerks Corporation of Austin, Texas
- Macintosh Programmer's Workshop (MPW) from Apple Computer, Inc. of Cupertino, CA

LabVIEW header files are compatible with these three environments. Header files may need modification for other environments.

CINs compiled for the 68K Macintosh will not be recognized by LabVIEW for the Power Macintosh, and vice versa.

LabVIEW does not currently work with fat binaries (a format that includes multiple executables in one file, in this case both 68K and Power Macintosh executables).

Microsoft Windows 3.1

LabVIEW for Windows supports external code compiled as a `.REX` file and prepared for LabVIEW using an application included with LabVIEW. This application requires `dos4gw.exe`, which comes with Watcom. LabVIEW is a 32-bit, flat memory-model application, so you must compile external code for a 32-bit memory model when you install the Watcom C compiler.

Watcom C is the only LabVIEW-supported compiler that can create 32-bit code of the correct format.

Microsoft Windows 95 and Windows NT

You can use CINs in LabVIEW for Windows 95/NT created with any of the following compilers.

- The Win32 Microsoft SDK (Software Developer's Kit) C/C++ command line compiler for Windows NT.

  See the *Microsoft Windows 95 and Windows NT* subsection of the *Compile the CIN Source Code* section of this chapter for information on how to create a CIN using this compiler.

- The Visual C++ for Windows NT C compiler.

  Use the same instructions as you would for the Microsoft C command line compiler. You also must add an `IDE=VC` line to the beginning of your `.lvm` file. See the *Microsoft Windows 95 and Windows NT* subsection of the *Compile the CIN Source Code* section of this chapter for information on how to create a CIN using this compiler.

- The Watcom C/386 compiler for Windows 3.1.

  With proper preparation, you can use CINs created using the Watcom C compiler for Windows 3.1 with LabVIEW for Windows 95/NT. See the *Microsoft Windows 95 and Windows NT* subsection of the *Compile the CIN Source Code* section of this chapter for more information on using the Watcom C compiler for Windows 3.1.

☞ **Note:** ***Under Windows 95 and Windows NT, you should not call CINs created using the Watcom complier that call DLLs and system functions or that access hardware directly. The technique Watcom uses to call such code under Windows 3.1 does not work under Windows 95 or Windows NT.***

Solaris

LabVIEW for the Sun supports external code compiled in `a.out` format under Solaris 1.*x* and a shared library format under Solaris 2.*x*. These formats are prepared for LabVIEW using a LabVIEW utility.

The unbundled Sun ANSI C compiler is the only compiler that has been tested thoroughly with LabVIEW. The header files are compatible with the unbundled Sun ANSI C Compiler and may need modification for other compilers.

HP-UX

LabVIEW for HP-UX supports external code compiled as a shared library. This library is prepared for LabVIEW using a LabVIEW utility.

The HP-UX C/ANSI C compiler is the only compiler that has been tested thoroughly with LabVIEW.

Steps for Creating a CIN

You create a CIN by first describing in LabVIEW the data you want to pass to the CIN. You then write the code for the CIN using one of the supported programming languages. After you compile the code, you run a utility on the compiled code that puts it into a format that LabVIEW can use. You then instruct LabVIEW to load the CIN.

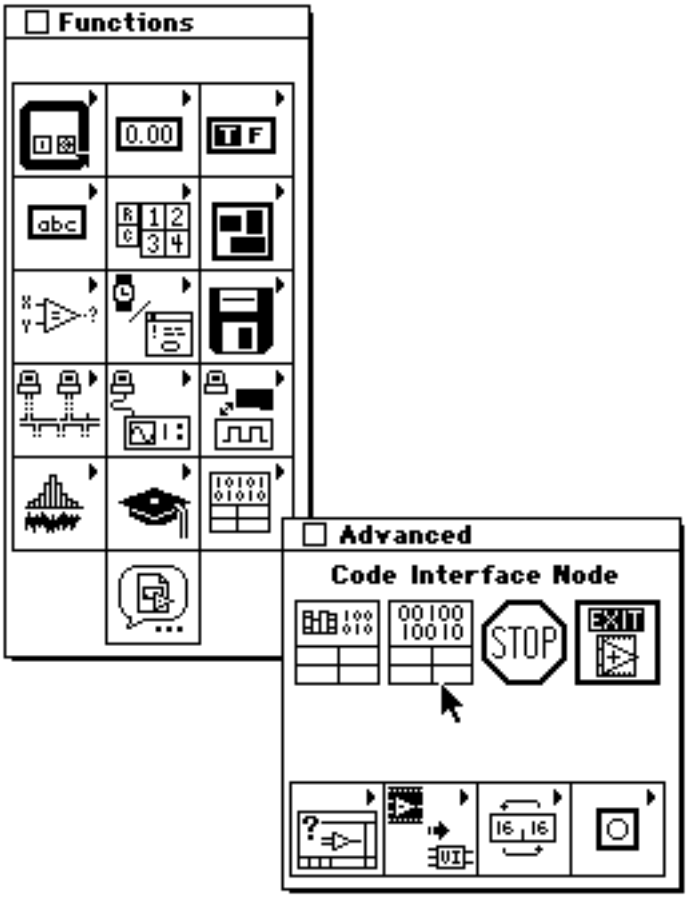
If you execute the VI at this point, and the block diagram needs to execute the CIN, LabVIEW calls the CIN object code and passes any data that is wired to the CIN. If you save the VI after loading the code, LabVIEW saves the CIN object code along with the VI so that LabVIEW no longer needs the original code to execute the CIN. You can update your CIN object code with new versions at any time.

The `examples` directory contains a `cins` directory that includes all of the examples given in this manual. The names of the directories in the `cins` directory correspond to the CIN name given in the examples.

The following steps explain how to create a CIN.

1. Place the CIN on a Block Diagram

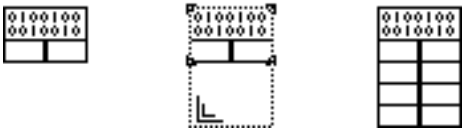
Select the Code Interface Node function from the **Advanced** palette of the **Functions** palette, as shown in the following illustration.



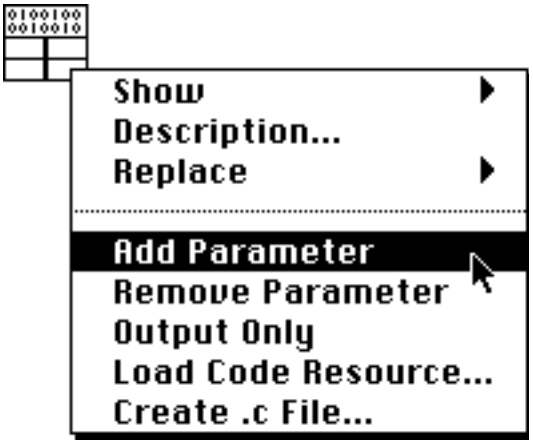
2. Add Input and Output Terminals to the CIN

A CIN has terminals with which you can indicate which data passes to and from a CIN. Initially, the CIN has one set of terminals, and you can pass a single value to and from the CIN. You add additional terminals by resizing the node or by selecting **Add Parameter** from the CIN terminal pop-up menu. Both methods are shown in the following illustration.

You can resize the node to add parameters,



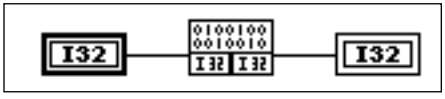
or use the pop-up menu to add a parameter.



Each pair of terminals corresponds to a parameter that LabVIEW passes to the CIN. The two types of terminal pairs are input-output and output-only.

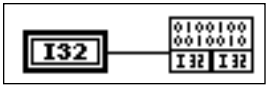
Input-Output Terminals

By default, a terminal pair is input-output; the left terminal is the input terminal, and the right terminal is the output terminal. As an example, consider a CIN that has a single terminal pair. Assume a 32-bit integer control is wired to the input terminal, and a 32-bit integer indicator is wired to the output terminal, as shown in the following illustration.



When the VI calls the CIN, the only argument LabVIEW passes to the CIN object code is a pointer to the value of the 32-bit integer input. When the CIN completes, LabVIEW then passes the value referenced by the pointer to the 32-bit integer indicator. When you wire controls and indicators to the input and the output terminals of a terminal pair, LabVIEW assumes that the CIN can modify the data passed. If another node on the block diagram needs the input value, LabVIEW may have to copy the input data before passing it to the CIN.

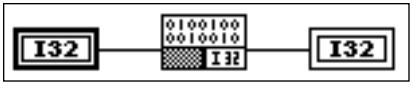
Now consider the same CIN, but with no indicator wired to the output terminal, as shown in the following illustration.



If you do not wire an indicator to the output terminal of a terminal pair, LabVIEW assumes that the CIN will not modify the value you pass to it. If another node on the block diagram uses the input data, LabVIEW does not copy the data. The source code should not modify the value passed into the input terminal of a terminal pair if you do not wire the output terminal. If the CIN does modify the input value, nodes connected to the input terminal wire may receive the modified data.

Output-Only Terminals

If you use a terminal pair only to return a value, make it an output-only terminal pair by selecting **Output Only** from the terminal pair pop-up menu. If a terminal pair is output-only, the input terminal is gray, as shown in the following illustration.



For output-only terminals, LabVIEW creates storage space for a return value and passes the value by reference to the CIN the same way that it passes values for input-output terminal pairs. If you do not wire a control to the left terminal, LabVIEW determines the type of the output parameter by checking the type of the indicator wired to the output terminal. This can be ambiguous if you wire the output to two

destinations that have different data types. You can remove this ambiguity by wiring a control to the left (input) terminal of the terminal pair as shown in the preceding figure. In this case, output terminal takes on the same data type as the input terminal. LabVIEW uses the input type only to determine the data type for the output terminal; the CIN does not use or affect the data of the input wire.

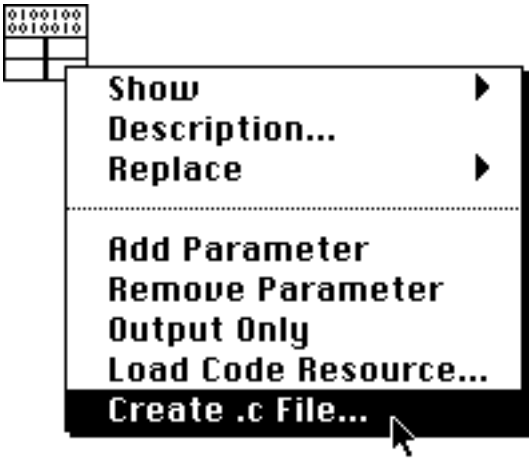
To remove a pair of terminals from a CIN, pop up on the terminal you want to remove and choose **Remove Terminal** from the menu. LabVIEW disconnects wires connected to the deleted terminal pair. Wires connected to terminal pairs below the deleted pair remain attached to those terminals and stretch to adjust to the terminals' new positions.

3. Wire the Inputs and Outputs to the CIN

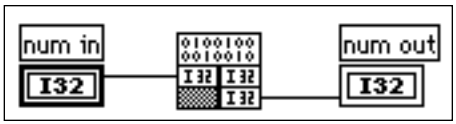
Connect wires to all the terminal pairs on the CIN to specify the data that you want to pass to the CIN, and the data that you want to receive from the CIN. The order of terminal pairs on the CIN corresponds to the order in which parameters are passed to the code. Notice that you can use any LabVIEW data types as CIN parameters. Thus, you can pass arbitrarily complex hierarchical data structures, such as arrays containing clusters which may in turn contain other arrays or clusters to a CIN. See Chapter 2, *CIN Parameter Passing*, for a description of how LabVIEW passes parameters of specific data types to CINs.

4. Create .c File

If you select **Create .c File...** from the CIN pop-up menu, as shown in the following illustration, LabVIEW creates a `.c` file in the style of the C programming language. The `.c` file describes the routines you must write and the data types for parameters that pass to the CIN.



For example, consider the following call to a CIN, which takes a 32-bit integer as an input and returns a 32-bit integer as an output.



The following code excerpt is the initial `.c` file for this node. It specifies seven routines (initially empty of any code) that must be written for the CIN. The `UseDefault` macros are the default for six of the seven routines, but because the `CINRun` routine is the most commonly used, it gets an actual function structure into which you can enter code. The six macros are actually `#define`s in `extcode.h`, which expand into empty routines; the `CINRun` routine is provided for you.

These seven routines are discussed in detail in subsequent chapters. The `.c` file is presented here to give you an idea of what LabVIEW creates at this stage in building a CIN.

```
/*
 * CIN source file
 */

#include "extcode.h"

/* stubs for advanced CIN functions */

UseDefaultCINInit
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave

CIN MgErr CINRun(int32 *num_in, int32 *num_out);

CIN MgErr CINRun(int32 *num_in, int32 *num_out) {

   /* ENTER YOUR CODE HERE */

   return noErr;
   }
```

This `.c` file is a template in which you must write C code. Notice that `extcode.h` is automatically included; it is a file that defines basic data types and a number of routines that can be used by CINs and external subroutines. `extcode.h` defines some constants and types whose definitions may conflict with the definitions of system header files. The LabVIEW `cintools` directory also contains a file, `hosttype.h`, that resolves these differences. This header file also includes many of the common header files for a given platform.

You should always use `#include "extcode.h"` at the beginning of your source code. If your code needs to make system calls, you should also use `#include "hosttype.h"` immediately after `#include "extcode.h"`, and then include your system header files. You should know that `hosttype.h` includes only a subset of the `.h` files for a given operating system. If the `.h` file you need is not included by `hosttype.h`, you can include it in the `.c` file for your CIN just after you include `hosttype.h`.

LabVIEW calls the `CINRun` routine when it is time for the node to execute. `CINRun` receives the input and output values as parameters.

The other routines (CINLoad, CINSave, CINUnload, CINAbort, CINInit, and CINDispose) are housekeeping routines, called at specific times to give you the opportunity to take care of specialized tasks with your CIN. For instance, CINLoad is called when a VI is first loaded. If you need to accomplish some special task when your VI is first loaded, put the code for that task in the CINLoad routine. To do this, first remove the UseDefaultCINLoad macro, and then write your CINLoad routine as follows:

```
CIN MgErr CINLoad(RsrcFile reserved) {
     Unused (reserved)
     /* your code goes here */
     return noErr;
     }
```

In general, you only need to write the CINRun routine. The other routines are mainly supplied for those instances in which you have special initialization needs, such as when your CIN must maintain some information across calls, and you want to preallocate or initialize global state information. The following code shows how to fill out the CINRun routine from the previously shown LabVIEW-generated .c file to multiply a number by two. This code is included for illustrative purposes. Chapter 2, *CIN Parameter Passing*, discusses in depth how LabVIEW passes data to a CIN, with several examples.

```
CIN MgErr CINRun(int32 *num_in, int32 *num_out) {
   *num_out = *num_in * 2;
   return noErr;
   }
```

Special Macintosh Considerations

If you compile your code for a 68K Macintosh, there are certain circumstances under which you must use the ENTERLVSB and LEAVELVSB macros at the entry and exit of some functions. These macros ensure that the global context register (A5 for MPW builds, A4 for Symantec/THINK and Metrowerks builds) for your CIN is established during your function, and that the caller's is saved and restored. This is necessary to enable you to reference global variables, call external subroutines, and call LabVIEW routines such as those described in subsequent chapters.

You need not use these macros in any of the seven predefined entry points (CINRun, CINLoad, CINUnload, CINSave, CINInit, CINDispose, CINAbort), because the CIN libraries already establish and restore the global context before and after calling these routines. Using them here would be harmless, but unnecessary.

However, if you create any other entry points to your CIN, you should use these macros. You create another entry point to your CIN whenever you pass the address of one of your functions to some other piece of code that may call your function later. An example of this would be in the use of the QSort routine in the LabVIEW support manager (described in the Online Reference or online manual). You must pass a comparison routine to QSort. This routine gets called directly by QSort, without going through the CIN library. Therefore it is your responsibility to set up your global context with ENTERLVSB and LEAVELVSB.

To use these macros properly, place the ENTERLVSB macro at the beginning of your function between your local variables and the first statement of the function. Place the LEAVELVSB macro at the end of your function just before returning, as in the following example.

```
CStr gNameTable[kNNames];

int32 MyComparisonProc(int32 *pa, int32 * pb)
  {
  int32 comparisonResult;

  ENTERLVSB

  comparisonResult = StrCmp(gNameTable[*pa],
  gNameTable[*pb]);

  LEAVELVSB
  return comparisonResult;
  }
```

The function MyComparisonProc is an example of a routine that might be passed to the QSort routine. Because it explicitly references a global variable (gNameTable), it must use the ENTERLVSB and LEAVELVSB macros. There are other things that can implicitly reference globals. Depending on the compiler and settings of various options, literal strings may also be referenced as globals.

It is best to always use the ENTERLVSB and LEAVELVSB macros whenever you create a new entry point to your CIN.

When you use these macros, be sure that your function does not return before calling the LEAVELVSB macro. One technique is to use a `goto endOfFunction` statement (where `endOfFunction` is a label just before the LEAVELVSB macro at the end of your function) in place of any return statements that you may place in your function.

5. Compile the CIN Source Code

You must compile the source code for the CIN in a format that LabVIEW can use. There are two steps to this process. First you compile the code using a compiler LabVIEW supports. Then you use a LabVIEW utility to modify the object code, putting it into a format that LabVIEW can use.

Because the compiling process is often complex, LabVIEW includes utilities that simplify the process. These utilities take a simple specification for a CIN and create object code you can load into LabVIEW. These tools vary depending on the platform and compiler you use. The following sections summarize the steps for each platform.

Note: ***Step 5 is different for each platform. Look under the heading for your platform and compiler in the following sections to find the instructions for your system.***

Every source code file for a CIN should list `#include "extcode.h"` ***before any other code. If your code needs to make system calls, you should also use*** `#include "hosttype.h"` ***immediately after*** `#include "extcode.h"`.

Macintosh

LabVIEW for the Macintosh uses external code as a customized code resource (on a 68K Macintosh) or as a shared library (on a Power Macintosh) that is prepared for LabVIEW using the separate utilities `lvsbutil.app` for THINK C or `lvsbutil.tool` for MPW. Both these utilities are included with LabVIEW.

You can create CINs on the Macintosh with compilers from any of the three major C compiler vendors: Symantec's THINK environment, Metrowerks' CodeWarrior environment, and Apple's Macintosh Programmer's Workshop (MPW) environment. Always use the latest

Universal headers that contain definitions for both 68K and Power Macintosh compilers.

The LabVIEW utilities for building Power Macintosh CINs are the same ones that are used for the 68K Macintosh and can be used to build both versions of a CIN. If you want to place both versions in the same folder, however, some development conflicts may arise. Because the naming conventions for object files and `.lsb` files are the same, make sure that one version does not replace the other. These kinds of issues can be handled in different ways, depending on your development environment.

Some CIN code that compiles and works on the 68K Macintosh and calls Macintosh OS or Toolbox functions may require changes to the source code before it will work on the Power Macintosh. Any code that passes a function pointer to a Mac OS or Toolbox function must be modified to pass a Routine Descriptor (see Apple's *Inside Macintosh* chapter on the Mixed Mode Manager, available in the Macintosh on RISC SDK from APDA). Also, if you use any 68K assembly language in your CIN, it must be ported to either C or Power Macintosh assembly language.

THINK C for 68K (Versions 5-7)

To create a THINK C CIN project, make a new folder for the project. Launch THINK C and create a new project in the new folder. The name of your THINK C project must match your CIN name exactly, and must not use any of the conventional project suffixes, such as `.π` or `.proj`. If you name your CIN `test`, your THINK C project must also be named `test`, so that it produces a link map file named `test.map`. You should keep the new project and the CIN files associated with it within the same folder.

With THINK C 7, an easy way to set up your CIN project is to make use of the project stationery in the `cintools:Symantec-THINK Files:Project Stationery` folder. For THINK C 7 projects the project stationery is a folder called `LabVIEW CIN TC7`. It provides a template for new CINs with almost all of the settings you need. See the `Read Me` file in the `Project Stationery` folder for details.

When building a CIN using THINK C for 68K, many of the preferences can be set to whatever you wish. Others, however, must be set to specific values to correctly create a CIN. If for some reason you do not

use the CIN project stationery, you will need to ensure that the following settings in the THINK C Preferences dialog box are made.

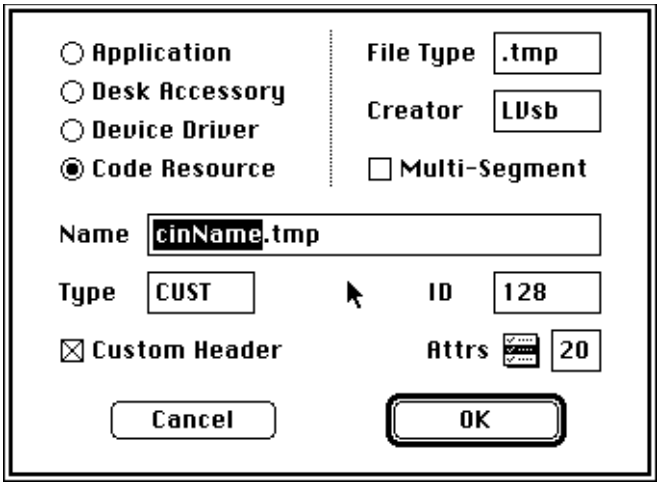
To set up your project if you are using THINK C 5, select **Options...** from the THINK C **Edit** menu. Then check the **Generate link map** box under **Preferences**.

If you are using THINK C 6 or 7, pull down the THINK C **Edit** menu and pop up on the **Options** item; select **THINK Project Manager...**. Under **Preferences**, check the **Generate link map** box, and then click on the **OK** button. Now go back to the **Options** item under the **Edit** menu and select **THINK C...**.

To complete the project set-up process for THINK C 5, THINK C 6, and THINK C 7, select the **Require prototypes** button under **Language Settings** and then check the **Check Pointer Types** box. Under **Prefix**, delete the line `#include <MacHeaders>` if it is present. Finally, under **Compiler Settings**, check the **Generate 68881 instructions** box, the **Native floating-point format** box, and the **Generate 68020 instructions** box. You can use the **Copy** button at the top of the dialog box to make these settings the default settings for new projects, which will make the set-up process for subsequent CINs simpler.

When you have finished selecting the options in the **Edit** menu, turn to the THINK C **Project** menu; select **Set Project Type...**. First, set the type to **Code Resource**. From the new options that appear, set the **File Type** to `.tmp`, the **Creator** to `LVsb`, the **Name** to the name of the CIN plus the extension `.tmp`, the **Type** to `CUST`, the **ID** to `128`, and check the **Custom Header** box. If you are creating a CIN called `test`, you

must name the resource `test.tmp`, as shown in the following illustration.

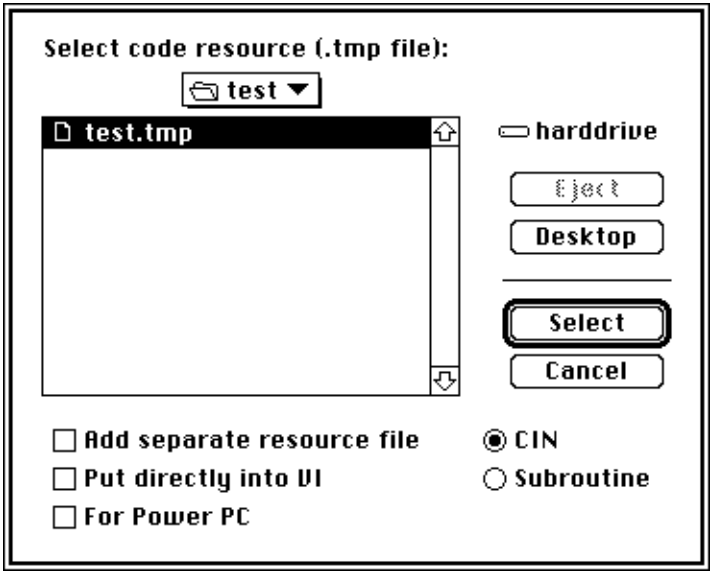


After these parameters are set, add both the `LVSBLib` and `CINLib` libraries (included with LabVIEW) to the project. Then add your `.c` files.

You are now ready to build the code resource. Go to the **Project** menu and select **Build Code Resource...**. A dialog box will appear, allowing you to save the code resource. The name of the code resource must be the same as the name of the CIN plus the extension `.tmp`.

After you build a code resource and give it a `.tmp` extension, you must run the application `lvsbutil.app`, also included with LabVIEW, to prepare external code for use as a CIN or external subroutine. This utility prompts you to select your `.tmp` file. The utility also uses the THINK C link map file, which carries a `.map` extension and must be in the folder with your `.tmp` file. The application `lvsbutil.app`

uses the .tmp and the .map files to produce a .lsb file that can be loaded into a VI.



If you are making a CIN, select the **CIN** option in the dialog box, as shown in the preceding illustration. If you are making a CIN for the Power Macintosh, also check the **For Power PC** box. If you are making an external subroutine, select the **Subroutine** option.

Advanced programmers can check the **Add separate resource file** box to add additional resources to their CINs or the **Put directly into VI** box to put the .lsb code into a VI without opening it or launching LabVIEW. Remember that the VI designated to receive the .lsb code must already contain .lsb code with the same name. Notice that you cannot put the code directly into a library.

If your .tmp code resource file does not show up in the dialog box, check its file type. When building the .tmp file, specify the file type as .tmp, which is under the **Set Project Type...** menu item of the **Project** menu in THINK C. The .lsb file this application produces is what the LabVIEW CIN node should load.

Note: ***The THINK C compiler will only find*** `extcode.h` ***if the file*** `extcode.h` ***is located on the THINK C search path. You can place the*** `cintools` ***folder in the same folder as your THINK C application, or you can make sure the line*** `#include "extcode.h"` ***is a full pathname to*** `extcode.h` ***under THINK C. For example:*** `#include "harddrive:cintools:extcode.h"`

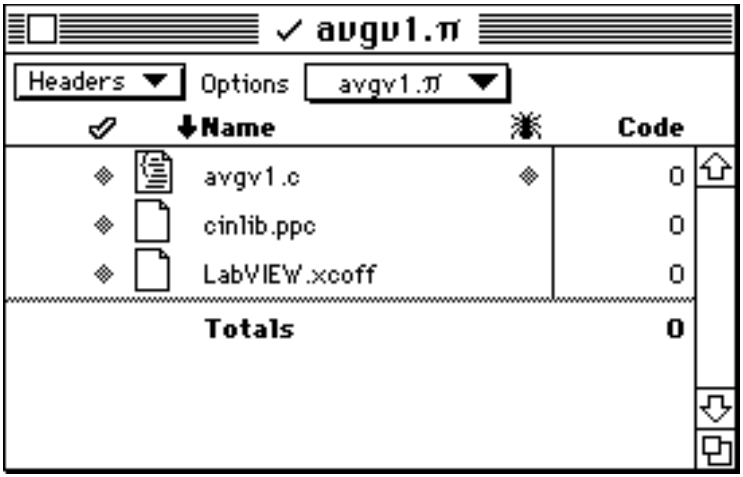
If you are using System 7.0 or later, you can extend the THINK C search path. To do so, first create a new folder in the same folder as your THINK C project and name it `Aliases`***. Then make an alias for the*** `cintools` ***folder, and drag this alias into your newly created*** `Aliases` ***folder. This technique enables the include line to read*** `#include "extcode.h"`***; therefore, it is not necessary to type the full pathname.***

Symantec C++ 8.0 for Power Macintosh

To create CINs using this environment, you will need to install the ToolServer application from the Symantec C++ 8.0 distribution disks. ToolServer is an Apple tool that performs the final linking steps in creating your CIN. It can be found in the `Apple Software:Tools` folder. Copy the `ToolServer 1.1.1` folder to your hard drive and place an alias to ToolServer in the `(Tools)` folder in your `Symantec C++ for PowerMac` folder.

You need the following files in your project to create a CIN for Power Macintosh.

- `CINLib.ppc`. This file is shipped with LabVIEW and can be found in the `cintools:PowerPC Libraries` folder.
- Your source files



You may also need the following file:

- `LabVIEW.xcoff`. This file is shipped with LabVIEW and can be found in the `cintools:PowerPC Libraries` folder. It is needed if you call any routines within LabVIEW e.g., `DSSetHandleSize()`, or `SetCINArraySize()`.

An easy way to set up your CIN project is to make use of the CIN project stationery in the `cintools:Symantec-THINK Files:Project Stationery` folder. For Symantec C version 8 projects the project stationery is a folder called `LabVIEW CIN SC8PPC`. The folder provides a template for new CINs containing almost all of the files and preference settings you need. See the `Read Me` file in the `Project Stationery` folder for details.

When building a CIN using Symantec C++ for PowerMac, many of the preferences can be set to whatever you wish. Others, however, must be set to specific values to correctly create a CIN. If for some reason you do not use the CIN project stationery, you will need to ensure that the following settings in the Symantec Project Manager **Options** dialog box (accessed from the **Project** menu) are made.

- Project Type—Set the **Project Type** pop-up menu to **Shared Library**. Set the **File Type** text field to `.tmp`. Set the **Destination** text field to *`cinName`*`.tmp`, where *`cinName`* is the name of your CIN. Set the **Creator** to `LVsb`.

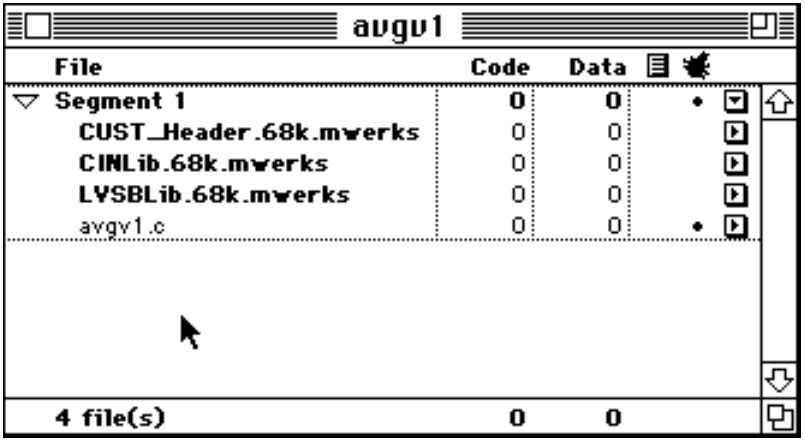
- Linker—Set the **Linker** pop-up menu to **PPCLink & MakePEF**. Set the **PPCLink settings** text field to `-export LVSBStart,LVSBhead`. Set the **MakePEF settings** text field to have `-l LabVIEW.xcoff.o=LabVIEW` in addition to the factory setting.
- Extensions—Set the **File Extension** text field to `.ppc`, the **Translator** pop-up menu to **XCOFF convertor**, and press the **Add** button.
- PowerPC C—In the **Compiler Settings** sub-page, select the **Align to 2 byte boundary** radio button. In the **Prefix** sub-page, remove the line that reads `#include <PPCMacheaders>`.

Build the CIN by selecting **Build Library** from the **Build** menu. Then convert the `.tmp` file with `lvsbutil.app` (with **For PowerPC** checked).

Metrowerks CodeWarrior for 68K

You need the following files in your project to be able to create a Metrowerks 68K CIN.

- `CustHdr.68k.mwerks` (This file *must* be the first file in the project.)
- `CINLib.68k.mwerks`
- `LVSBLib.68k.mwerks`
- Your source files



Note: ***All of your files must be in a single segment. LabVIEW does not support multi-segment CINs.***

An easy way to set up your CIN project is to make use of the CIN project stationery in the `cintools:Metrowerks Files:Project Stationery` folder. For CodeWarrior 68K projects the project stationery is a file called `LabVIEW CIN MW68K`. The file provides a template for CINs containing almost all of the settings you need. See the `Read Me` file in the `Project Stationery` folder for details.

When building a CIN using CodeWarrior for 68K, many of the preferences can be set to whatever you wish. Others, however, must be set to specific values to correctly create a CIN. If for some reason you do not use the CIN project stationery, you will need to ensure that the following settings in the CodeWarrior Preferences dialog box are made.

- Language—Set the **Source Model** pop-up menu to **Apple C**. Empty the **Prefix File** text field.
- Processor—Check the **68881 Codegen** and **MPW C Calling Conventions** checkboxes. Leave the **4-Byte Ints** and **8-Byte Doubles** checkboxes unchecked.
- Linker—Check the **Generate Link Map** checkbox.
- Project—Set the **Project Type** pop-up menu to **Code Resource**. Set the **File Name** text field to *`cinName`*`.tmp`, where *`cinName`* is the name of your CIN. Set the **Resource Name** text field to the same text as in the **File Name** text field. Set the **Type** text field to `.tmp` and the **ResType** text field to `CUST`. Set the **ResID** text field to `128`. Set the **Header Type** pop-up menu to **Custom**. Set the **Creator** to `LVsb`.
- Access Paths—Add your `cintools` folder to the list of access paths.

Build the CIN by selecting **Make** from the CodeWarrior **Project** menu.

Caution: ***This operation will destroy the contents of any other file named*** *`cinName`*`.tmp` ***in that folder. This could easily be the case if this is the same folder in which you build a Power Macintosh version of your CIN. If you are building for both platforms, you should keep separate directories for each. The convention used by the MPW CIN tools is to have two subdirectories named*** `PPCObj` ***and*** `M68Obj` ***where all platform-specific files reside.***

Note: ***If you have both a ThinkC68K and a MetrowerksC68K map file, lbsbutil cannot know in advance which compiler your*** `.tmp` ***file came from. It will first look for a ThinkC*** `.map` ***file, then for a Metrowerks*** `.map` ***file. To avoid any conflict, remove the unnecessary*** `.map` ***file before using*** `lvsbutil.app`***.***

When you have successfully built the *`cinName`*`.tmp` file, you must then use the `lvsbutil.app` application to create the *`cinName`*`.lsb` file.

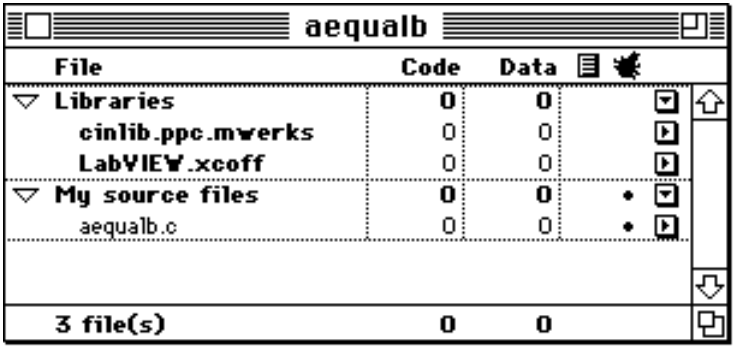
The `lvsbutil.app` application has a checkbox in the file selection dialog box labelled **For Power PC**. This checkbox *must not* be checked for 68K CINs. Select any other options that you want for your CIN, and then select your *`cinName`*`.tmp` file. *`cinName`*`.lsb` will be created in the same folder as *`cinName`*`.tmp`.

Caution: ***This operation will destroy the contents of any previous file named*** `cinName.lsb` ***in that folder. This could easily be the case if this is the same folder in which you build a 68K Macintosh version of your CIN.***

Metrowerks CodeWarrior for Power Macintosh

You need the following files in your CodeWarrior project to create a CIN for Power Macintosh.

- `CINLib.ppc.mwerks`. This file is shipped with LabVIEW and can be found in the `cintools:Metrowerks Files:68K Libraries` folder.
- Your source files



You may also need the `LabVIEW.xcoff` file. This file is shipped with LabVIEW and can be found in the `cintools:PowerPC`

Libraries folder. It is needed if you call any routines within LabVIEW e.g., DSSetHandleSize(), or SetCINArraySize().

Finally, if you call any routines from a system shared library, you must add the appropriate shared library interface file to your project's file list.

An easy way to set up your CIN project is to make use of the CIN project stationery in the cintools:Metrowerks Files:Project Stationery folder. For CodeWarrior PowerPC projects the project stationery is a file called LabVIEW CIN MWPPC. This file provides a template for CINs containing almost all of the settings you need. See the Read Me file in the Project Stationery folder for details.

When building a CIN using CodeWarrior for PPC, many of the preferences can be set to whatever you wish. Others, however, must be set to specific values to correctly create a CIN. If for some reason you do not use the CIN project stationery, you will need to ensure that the following settings in the CodeWarrior Preferences dialog box are made.

- Language—Set the **Source Model** pop-up menu to **Apple C**. Empty out the **Prefix File** text field (using **MacHeaders** will not work).
- Processor—Set the **Struct Alignment** pop-up menu to 68K.
- Linker—Empty all of the **Entry Point** fields.
- PEF—Set the **Export Symbols** pop-up menu to **Use .exp file** and place a copy of the file projectName.exp (found in your cintools:Metrowerks Files:PPC Libraries folder) in the same folder as your CodeWarrior project. Rename this file to *projectName*.exp, where *projectName* is the name of the project file. CodeWarrior will look in this file to determine what symbols your CIN exports. LabVIEW needs these to link to your CIN.
- Project—Set the **Project Type** pop-up menu to **Shared Library**. Set the file name to be *cinName*.tmp, where *cinName* is the name of your CIN. Set the **Type** field to .tmp. Set the **Creator** to LVsb.
- Access Paths—Add your cintools folder to the list of access paths.

Build the CIN by selecting **Make** from the CodeWarrior **Project** menu.

Caution: ***This operation will destroy the contents of any other file named*** *cinName*`.tmp` ***in that folder. This could easily be the case if this is the same folder in which you build a 68K Macintosh version of your CIN. If you are building for both platforms, you should keep separate folders for each. The convention used by the MPW CIN tools is to have two subdirectories named*** `PPCObj` ***and*** `M68Obj` ***where all platform-specific files reside.***

When you have successfully built the *cinName*`.tmp` file, you must then use the `lvsbutil.app` application to create the *cinName*`.lsb` file.

The `lvsbutil.app` application has a checkbox in the file selection dialog box labelled **For Power PC**. Check this box, along with any other options that are necessary for your CIN, and then select your *cinName*`.tmp` file. *cinName*`.lsb` will be created in the same folder as *cinName*`.tmp`.

Caution: ***This operation will destroy the contents of any previous file named*** *cinName*`.lsb` ***in that folder. This could easily be the case if this is the same folder in which you build a 68K Macintosh version of your CIN.***

Macintosh Programmer's Workshop for 68K and Power Macintosh

Macintosh Programmer's Workshop (MPW) can be used to build CINs for either the Motorola 680*x*0 (68K) Macintosh or the Power Macintosh. Several scripts are available for the MPW environment to help you build CINs. To deal with the problem of building CINs for two different CPUs, these new scripts are designed to use two subdirectories in your CIN folder: `PPCObj` and `M68Obj`. The platform-specific object and CIN files are kept in these subdirectories. The scripts make use of the first generation PowerPC C compiler from Apple, `PPCC`, and the standard 68K C compiler, `C`. Although newer compilers (`MrC` and `SC`) are in beta test as this publication goes to press, the scripts have not yet been updated to use them. The scripts are:

- `CINMake`—A script capable of building both Power Macintosh and 68K Macintosh CINs. It uses a simplified form of a makefile that you provide. It can be run every time you need to rebuild your CIN.
- `LVMakeMake`—A script that is similar to the `lvmkmf` (LabVIEW Make Makefile) script available for building CINs under the

Solaris operating system. It builds a skeletal but complete makefile that you can then customize and use with the MPW `make` tool.

`CINMake` can be used for building both Power Macintosh and 68K Macintosh versions of your CINs. By default, the `CINMake` script builds 68K Macintosh CINs and puts the resulting *`cinName`*`.lsb` into the `M68Obj` folder.

You must have one makefile for each CIN. Name the makefile by appending `.lvm` to the CIN name. This indicates that this is a LabVIEW makefile. The makefile should resemble the following pseudocode. Be sure that each `Dir` command ends with the colon character (`:`).

`name = `*`name`* — Name for the code; indicates the base name for your CIN. The source code for your CIN should be in *`name`*`.c`. The code created by the makefile is placed in a new file, *`name`*`.lsb` (`.lsb` is a mnemonic for LabVIEW subroutine).

`type = `*`type`* — Type of external code you want to create. For CINs, you should use a type of `CIN`.

`codeDir = `*`codeDir:`* — Complete pathname to the folder containing the `.c` file used for the CIN.

`cinToolsDir = `*`cinToolsDir`*`:` — Complete pathname to the LabVIEW `cintools:MPW` folder, which is located in the LabVIEW folder.

`LVMVers = 2` — Version of `CINMake` script reading this `.lvm` file.

`inclDir = -i `*`inclDir`*`:` — (optional) Complete or partial pathname to a folder containing any additional `.h` files.

`otherM68ObjFiles = otherM68ObjFiles`
(optional) For 68K Macintosh only, list of additional object files (files with a `.o` extension) that your code needs to compile. Separate the names of files with spaces.

`otherPPCObjFiles = otherPPCObjFiles`
(optional) For Power Macintosh only, list of additional object files (files with a `.o` extension) that your code needs to compile. Separate the names of files with spaces.

`subrNames = subrNames` (optional) For 68K Macintosh only, list of external subroutines the CIN calls. You need `subrNames` only if the CIN calls external subroutines. Separate the names of subroutines with spaces.

`ShLibs = sharedLibraryNames`
(optional) For Power Macintosh only, a space-separated list of the link-time copies of import libraries with which the CIN must be linked. Each should be a complete path to the file.

`ShLibMaps = sharedLibMappings`
(optional) For Power Macintosh only, the command-line arguments to the `MakePEF` tool that indicate the mapping between the name of each link-time import library and the run-time name of that import library. These will usually look something like the following:

```
"-l libA.xcoff=libA
   -l libB.xcoff=libB"
```

Notice that only the file names are needed, not entire paths.

You must adjust the `–Dir` names to reflect your own file system hierarchy.

Modify your MPW command search path by appending the cintools:MPW folder to the default search path. This search path is defined by the MPW Shell variable commands.

```
set commands "{commands}","<pathname to directory of cinToolsDir>"
```

Go to the MPW Worksheet and enter the following two commands. First, set your current folder to the CIN folder using the MPW Directory command:

```
Directory <pathname to directory of your CIN>
```

Next, run the LabVIEW CINMake script:

```
CINMake <name of your CIN>
```

If CINMake does not find a .lvm file in the current folder, it will build a file named *cinName*.lvm, and prompt you for necessary information. This file, *cinName*.lvm, will be in a format compatible with building both Power Macintosh and 68K Macintosh CINs in the same folder. If CINMake finds a *cinName*.lvm but it does not have the line LVMVers = 2, it will save the .lvm file in *cinName*.lvm.old and update the *cinName*.lvm file to be compatible with the new version of CINMake.

You can use LVMakeMake to build an MPW makefile that you can then customize for your own purposes. You should only have to run LVMakeMake once for a given CIN. You may then want to modify the resulting makefile by adding the proper header file dependencies, or by adding other object files to be linked into your CIN. The format of a LVMakeMake command follows, with optional parameters listed in brackets.

```
LVMakeMake [-o makeFile] [-PPC] <name of your CIN>.make
```

-o	makeFile specifies the name of the output makefile. If this argument is not specified, LVMakeMake writes to standard output.
-PPC	If this argument is specified, a makefile suitable for building a Power Macintosh CIN is created. By default, a 68K Macintosh makefile is created.

For example, to build a Power Macintosh makefile for a CIN named myCIN, execute the following command:

```
LVMakeMake -PPC myCIN > myCIN.ppc.make
## creates the makefile
```

You can then use the MPW `make` tool to build your CIN, as shown in the following commands.

```
make -f myCIN.ppc.make> myCIN.makeout
## creates the build commands
myCIN.makeout
## executes the build commands
```

You should load the `.lsb` file this application produces into your LabVIEW CIN node.

Microsoft Windows 3.*x*

Microsoft Windows 3.*x* is a 16-bit operating system. Most applications written for it are 16-bit applications. A 16-bit application faces several obstacles when working with large amounts of information, such as manipulating arrays that require more than 64 kilobytes of memory.

LabVIEW is a 32-bit application without most of the inherent limitations found in 16-bit applications. Because of the way that CINs are linked to VIs, however, LabVIEW can use only code compiled for 32-bit applications. This is because CINs reside in the same memory space as VIs and work with LabVIEW data. To create CINs, a compiler must be able to create 32-bit relocatable object code.

The only compiler that currently supports the correct format of executables is Watcom C. The following section lists the steps for compiling a CIN with the Watcom compiler.

Watcom C Compiler

With the Watcom C compiler, you create a specification that includes the name of the file you want to create, relevant directories, and any external subroutines or object files the CIN needs. (External subroutines are described in Chapter 4, *External Subroutines*.) You then use the `wmake` utility included with Watcom to compile the CIN.

In addition to compiling the CIN, the makefile directs `wmake` to put the CIN in the appropriate form for LabVIEW.

The makefile should look like the following pseudocode. You should append `.lvm` to the makefile name to indicate that this is a LabVIEW makefile.

`name = name`	Name for the code; indicates the base name for your CIN. The source code for your CIN should be in *name*`.c`. The code created by the makefile is placed in a new file, *name*`.lsb` (`.lsb` is a mnemonic for LabVIEW subroutine).
`type = type`	Type of external code you want to create. For CINs, you should use a type of `CIN`.
`codeDir = codeDir`	Complete or partial pathname to the directory containing the `.c` file used for the CIN.
`wcDir = wcDir`	Complete or partial pathname to the directory containing the Watcom compiler.
`CinToolsDir = CinToolsDir`	Complete or partial pathname to the LabVIEW `cintools` directory, which is located in the LabVIEW directory. This directory contains header files you can use for creating CINs, and tools that the `wmake` utility uses to create the CIN.
`inclDir = inclDir`	(optional) Complete or partial pathname to a directory containing any additional `.h` files.
`objFiles = objFiles`	(optional) List of additional object files (files with an `.obj` extension) that your code needs to compile. Separate the names of files with spaces.

`subrNames = subrNames` (optional) List of external subroutines the CIN calls. You need `subrNames` only if the CIN calls external subroutines. Separate the names of subroutines with spaces.

```
!include $(CinToolsDir)\generic.mak
```

Execute the `wmake` command by entering the following in DOS.

```
wmake /f <name of your CIN>.lvm
```

Note: ***The*** `wmake` ***utility sometimes erroneously stops a make with an incorrectly reported error when it is run in the DOS shell within Windows. If this happens, run it in normal DOS.***

The `wmake` utility scans the specified LabVIEW makefile and remembers the defined values. The last line of the makefile, `!include $(CinToolsDir)\generic.mak`, instructs `wmake` to compile the code resource based on instructions in the `generic.mak` file, which is stored in the `cintools` directory. The `wmake` utility compiles the code and then transforms it into a form that LabVIEW can use. The resulting code is stored in a *`name`*`.lsb` file, where *`name`* is the CIN name given in the `name` line of the makefile.

Note: ***You cannot link most of the Watcom C libraries to your CIN because precompiled libraries contain code that cannot be properly resolved by LabVIEW when it links a VI to a CIN. If you try to call those functions, your CIN may crash.***

LabVIEW provides functions that correspond to many of the functions in these libraries. These functions are described in subsequent chapters of this manual. If you need to call a function that is not supplied by LabVIEW, you can access the function from a dynamic link library (DLL). A CIN can call a DLL using the techniques described in the Watcom C manuals. A DLL can call any function from the C libraries. See Chapter 3, *CIN Advanced Topics****, for information on calling a DLL.***

Microsoft Windows 95 and Windows NT

You can use the Microsoft SDK C/C++ or the Visual C++ compiler to build CINs for LabVIEW for Windows 95/NT. With proper preparation, you can also use some CINs created using Watcom C for Windows 3.1.

Microsoft SDK C/C++ Compiler

The method for building CINs under Windows 95 or Windows NT is similar to the method for building CINs under Windows 3.1 using the Watcom C compiler.

1. Add a `CINTOOLSDIR` definition to your list of user environment variables.

   Under Windows NT, you can edit this list with the `System` control panel accessory. For example, if you installed LabVIEW for Windows 95/NT in `c:\lv31nt`, the CIN tools directory should be `c:\lv31nt\cintools`. In this instance, you would add the following line to the user environment variables using the System control panel.

   ```
   CINTOOLSDIR = c:\lv31nt\cintools
   ```

   Under Windows 95, you must modify your AUTOEXEC.BAT, to set CINTOOLSDIR to the correct value.

2. Build a `.lvm` file (LabVIEW Makefile) for your CIN. LabVIEW for Windows 95/NT requires you to define fewer variables than LabVIEW for Windows 3.1. You must specify the items in the following list.

   - name = name of CIN or external subroutine (mult, for example)
   - type = CIN or LVSB (depending on whether it is a CIN or an external subroutine)
   - `!include $(CINTOOLSDIR)\ntlvsb.mak`

   If your CIN uses extra object files or external subroutines, you can specify the `objFiles` and `subrNames` options. You do not need to specify the `codeDir` parameter, because the code for the CIN must be in the same directory as the makefile. You do not need to specify the `wcDir` parameter, because the CIN tools can determine the compiler's location.

   You can compile the CIN code using the following command, where `mult` is the makefile name.

   ```
   nmake /f mult.lvm
   ```

If you want to use standard C or Windows 95 or Windows NT libraries, define the symbol `cinLibraries`. For example, to use standard C functions in the preceding example, you could use the following `.lvm` file.

```
name = mult
type = CIN
cinLibraries=libc.lib
!include $(CINTOOLSDIR)\ntlvsb.mak
```

To include multiple libraries, separate the list of library names using spaces.

Visual C++ for Windows 95 or Windows NT

To build CINs under Windows NT or Windows 95 using Visual C++, follow the instructions for the Microsoft SDK C/C++ compiler, listed in the preceding section. The one difference is that you must add an `IDE = VC` line to the beginning of your `.lvm` file.

Watcom C Compiler for Windows 3.1 under Windows 95 or Windows NT

CINs you have created using the Watcom C compiler for Windows 3.1 should work under Windows 95 or Windows NT. However, if your CIN makes calls to communicate with hardware drivers, performs register or memory mapped I/O, or calls Windows 3.1 functions, it may not work without modification. Windows 3.1 drivers do not run under Windows 95 or Windows NT, so you will have to port any drivers that you may have written for Windows 3.1 to Windows 95 or Windows NT. Also, CINs cannot manipulate hardware directly. To perform register or memory-mapped I/O, you will have to write a Windows 95 or Windows NT driver. If you call Windows 3.1 functions, you should check to make sure that those functions are still valid under Windows 95 and Windows NT.

To create CINs using Watcom C for Windows 3.1, follow the Watcom C instructions given in the *Watcom C Compiler* subsection of the *Compile the CIN Source Code* section of this chapter. You must compile the source code for the CINs under Windows 3.1. Use the LabVIEW for Windows 3.1 CIN libraries to compile the CINs.

Solaris 1.*x*

LabVIEW for the Sun can use external code compiled in `a.out` format and prepared for LabVIEW using a LabVIEW utility. The unbundled Sun C compiler is the only compiler that has been tested thoroughly with LabVIEW. Other compilers that can generate code in `a.out` format might also work with LabVIEW, but this has not been verified. The C compiler that comes with the Sun does not use extended-precision floating-point numbers; code using this numeric type will not compile. However, the unbundled C compiler does use them.

Solaris 2.*x*

The preceding information for Solaris 1.*x* is true for Solaris 2.*x*, with one exception—LabVIEW 3.1 and higher for Solaris 2.*x* uses code compiled in a shared library format, rather than the `a.out` format previously specified.

☞ **Note:** ***LabVIEW 3.0 for Solaris 2.x supported external code compiled in ELF format.***

Existing Solaris 1.*x* and Solaris 2.*x* (for LabVIEW 3.0) CINs will not operate correctly if they reference functions that are not in the System V Interface Definition (SVID) for `libc`, `libsys`, and `libnsl`. Recompiling your existing CINs using the shared library format should ensure that your CINs function as expected.

HP-UX

As previously stated, the HP-UX C/ANSI C compiler is the only compiler that has been tested with LabVIEW.

Unbundled Sun ANSI C Compiler and HP-UX C/ANSI C Compiler

With these compilers, you create a makefile using the shell script `lvmkmf` (LabVIEW Make Makefile), which creates a makefile for a given CIN. You then use the standard `make` command to make the CIN code. In addition to compiling the CIN, the makefile puts the code in a form that LabVIEW can use.

The format for the `lvmkmf` command follows, with optional parameters listed in brackets.

```
lvmkmf [-o Makefile] [-t CIN] [-ext Gluefile] LVSBName
```

LVSBName, the name of the CIN or external subroutine that you want to build, is required. If LVSBName is foo, the compiler assumes the source is foo.c, and the compiler names the output file foo.lsb.

-o is optional and supplies the name of the makefile that lvmkmf creates. If you do not use this option, the makefile name defaults to Makefile.

-t is optional and indicates the type of external code you want to create. For CINs, you should use CIN, which is the default.

-ext is needed only if this external code calls external subroutines. The argument to this directive is the name of a file that contains the names of all subroutines that this code calls, with one name per line. The file is not necessary to run the lvmkmf script, but it must be present before you can successfully make the CIN. If you do not specify a -ext option, lvmkmf assumes that the CIN does not reference any external subroutines.

In Solaris 1.*x*, the makefile produced assumes that the directories for the files cin.o, cinetc.o, makeglueBSD.awk, and lvsbutil are in certain locations. If these assumptions are incorrect, you can edit the makefile to correct the pathnames.

In Solaris 2.*x*, the makefile produced assumes that the directories for the files cin.o, cinetc.o, makeglueSVR4.awk, and lvsbutil are in certain locations. If these assumptions are incorrect, you can edit the makefile to correct the pathnames.

In HP-UX, the makefile produced assumes that the directories for the files cin.o, cinetc.o, makeglueHP.awk, and lvsbutil are in certain locations. If these assumptions are incorrect, you can edit the makefile to correct the pathnames.

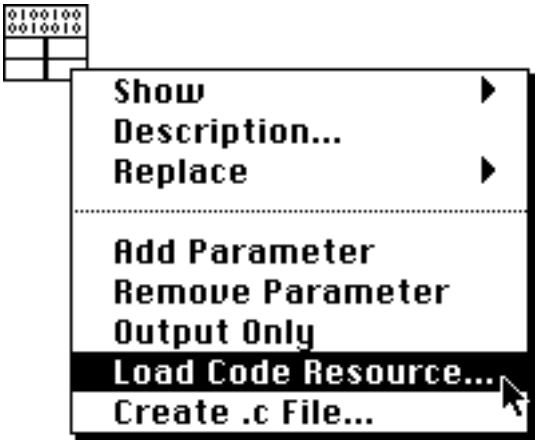
If you specify the -ext argument to the lvmkmf script, the makefile creates temporary files. For example, if the gluefile name is bar, the makefile creates files bar.s and bar.o. Neither the CIN nor the makefile needs these files after the CIN has been created.

If you make external subroutines, you need to create a separate makefile for them. The lvmkmf script creates a file called Makefile unless you use the -o option. For this reason, you may want to place the code for each subroutine in separate directories to avoid writing over one Makefile with the other. If you want to place the code in

the same directory, you need either to combine the two makefiles manually, or to create two separate makefiles (using the `-o` option to the `lvmkmf` script) and use `make -f <makefile>` to create the CIN or external subroutine.

6. Load the CIN Object Code

Load the code resource by choosing **Load Code Resource** from the CIN pop-up menu. Select the `.lsb` file you created in step 5, *Compile the CIN Source Code*.



This command loads your object code into memory and links the code to the current front panel/block diagram. After you save the VI, the file containing the object code does not need to be resident on the computer running LabVIEW for the VI to execute.

If you make modifications to the source code, you can load the new version of the object code using the **Load Code Resource** option. The file containing the object code for the CIN must have an extension of `.lsb`.

There is no limit to the number of CINs per block diagram.

LabVIEW Manager Routines

LabVIEW has a suite of routines that can be called from CINs and external subroutines. This suite of routines performs user-specified

routines using the appropriate instructions for a given platform. These routines, which manage the functions of a specific operating system, are grouped into three categories, the memory manager, the file manager, and the support manager.

External code written using the managers is portable–you can compile it without modification on any platform that supports LabVIEW. This portability has two advantages. First, the LabVIEW application is built on top of the managers; except for the managers, the source code for LabVIEW is identical across platforms. Second, the analysis VIs are built mainly from CINs; the source code for these CINs is the same for all platforms.

For general information about the memory manager, the file manager, and the support manager, see Chapter 5, *Manager Overview*.

Online Reference

For desciptions of the functions, or of the file manager data structures, select **Online Reference** from LabVIEW's **Help** menu. Click on the topic, *Function and VI Reference*, and then the relevant subtopic. Or see the *Code Interface Node Reference* online manual.

Pointers as Parameters

Some manager functions have a parameter that is a *pointer*. These parameter type descriptions are identified by a trailing asterisk (such as the **hp** parameter of the `AZHandToHand` memory manager function documented in the Online Reference) or are type defined as such (such as the **name** parameter of the `FNamePtr` function documented in the Online Reference). In most cases, this means the manager function will write a value to pre-allocated memory. In some cases, such as `FStrFitsPath` or `GetALong`, the function reads a value from the memory location, so you don't have to pre-allocate memory for a return value.

Table 1-1 lists the functions with parameters that return a value for which you must pre-allocate memory.

Table 1-1. Functions with Parameters Needing Pre-allocated Memory

AZHandToHand	FGetInfo	FPathToDString
AZMemStats	FGetPathType	FPathToPath
AZPtrToHand	FGetVolInfo	FRefNumToFD
DateToSecs	FMOpen	FStringToPath
DSHandToHand	FMRead	FTextToPath
DSMemStats	FMTell	FUnflattenPath
DSPtrToHand	FMWrite	GetAlong
FCreate	FNamePtr	NumericArrayResize
FCreateAlways	FNewRefNum	RandomGen
FFlattenPath	FPathToArr	SecsToDate
FGetAccessRights	FPathToAZString	SetALong
FGetEOF		

It is important to actually allocate space for this return value. The following examples illustrate correct and incorrect ways to call one of these functions from within a generic function `foo`:

Correct example:

```
foo(Path path) {
  Str255 buf; /* allocated buffer of 256 chars */
  File fd;
  MgErr err;

  err = FNamePtr(path, buf);
  err = FMOpen(&fd, path, openReadOnly,
              denyWriteOnly);
  }
```

Incorrect example:

```
foo(Path path) {
  PStr p;     /* an uninitialized pointer */
  File *fd;   /* an uninitialized pointer */
  MgErr err;

  err = FNamePtr(path, p);
  err = FMOpen(fd, path, openReadOnly,
               denyWriteOnly);
  }
```

In the correct example, `buf` contains space for the maximum-sized Pascal string (whose address is passed to `FNamePtr`), and `fd` is a local variable (allocated space) for a file descriptor.

In the incorrect example, `p` is a pointer to a Pascal string, but the pointer is not initialized to point to any allocated buffer. `FNamePtr` expects its caller to pass a pointer to an allocated space, and writes the name of the file referred to by `path` into that space. Even if the pointer does not point to a valid place, `FNamePtr` will write its results there, with unpredictable consequences. Similarly, `FMOpen` will write its results to the space to which `fd` points, which is not a valid place because `fd` is uninitialized.

Debugging External Code

LabVIEW has a debugging window that you can use with external code to display information at execution time. You can open the window, display arbitrary print statements, and close the window from any CIN or external subroutine.

Use the `DbgPrintf` function to create this debugging window. The format for `DbgPrintf` is similar to the format of the `SPrintf` function, which is described in the Online Reference. `DbgPrintf` takes a variable number of arguments, where the first argument is a C format string.

DbgPrintf

syntax `int32 DbgPrintf(CStr cfmt, ..);`

The first time you call `DbgPrintf`, LabVIEW opens a window to display the text you pass the function. Subsequent calls to `DbgPrintf` append new data as new lines in the window (you do not need to pass in the new line character to the function). If you call `DbgPrintf` with `NULL` instead of a format string, LabVIEW closes the debugging window. You cannot position or change the size of the window.

The following examples show how to use `DbgPrintf`.

```
DbgPrintf("");          /* print an empty line,
                        opening the window if
                        necessary */

DbgPrintf("%H", var1); /* print the contents of an
                        LStrHandle (LabVIEW string),
                        opening the window if
                        necessary */

DbgPrintf(NULL);        /* close the debugging window
                        */
```

Debugging CINs Under Windows 95 and Windows NT

The Windows 95 and Windows NT platforms support source level debugging of CINs. To enable debugging, add the following line to the `.lvm` file of your CIN:

```
DEBUG = 1

cinLibraries = Kernel32.lib
```

These lines will add debug information to the CIN. You must also add the `DebugBreak` system call at the point where you want to break into the CIN code. After adding this information, recompile the CIN and reload it into LabVIEW. When you run the VI containing the CIN, a dialog box will appear with the following message:

```
A Breakpoint has been reached. Click OK to
terminate application. Click CANCEL to debug the
application.
```

Click **CANCEL**. This launches the debugger, which attaches to LabVIEW, searches for the DLLs, and then asks for the source file of

your CIN. Point to your source file and the debugger will load the CIN source code. You can then proceed to debug your code.

Debugging CINs Under Sun or Solaris

It is not currently possible to use Sun's debugger, `dbx`, to debug CINs. The best you can do is use standard C `printf` calls or the `DbgPrintf` function mentioned earlier.

Debugging CINs Under HP-UX

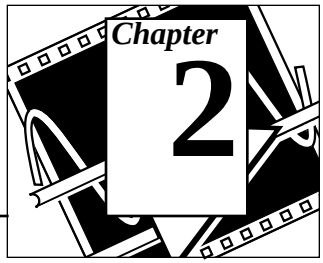
You can debug CINs built on the HP-UX platform using `xdb`, the HP source level debugger. To do so, compile the CIN with debugging turned on. You must also enable shared library debugging with the `-s` flag and direct `xdb` to the source files for your CIN. For example, if your CIN source code is in the `tests/first` directory, you could invoke `xdb` with the following command:

```
xdb -s -d tests/first labview
```

See the `xdb` manual for more information. Once the CIN is loaded, break into the debugger and set your breakpoints. You may need to qualify function names with the name of the shared library. Qualified names are in the form `function_name@library_name`. The name of the shared library will not be what it was when compiled. Instead, it will be a unique name generated by the C library function `tmpnam`. The name will always begin with the string `LV`. Use the debugger command `mm` to display the memory map of all currently loaded shared libraries. CIN shared libraries are ordered by load time on the name space, so CINs loaded later appear in the memory map before CINs loaded earlier. As an example, to break at `CINRun` for the library `/usr/tmp/LVAAAa17732`, set the breakpoint as follows:

```
>b CINRun@LVAAAa17732
```

If you reload a CIN that is already loaded, the debugger will not function properly. If you change a CIN, you must quit and restart the debugger to enable it to work as desired.



CIN Parameter Passing

This chapter describes the data structures that LabVIEW uses when passing data to a CIN.

Introduction

LabVIEW passes parameters to the `CINRun` routine. These parameters correspond to each of the wires connected to the CIN. You can pass any data type to a CIN that you can construct in LabVIEW; there is no limit to the number of parameters you can pass to and from the CIN.

CIN .c File

When you select the **Create .c File...** option, LabVIEW creates a `.c` file in which you can enter your CIN code. The `CINRun` function and its prototype are given, and its parameters are typed to correspond to the data types being passed to the CIN in the block diagram. If you want to refer to any of the other six CIN routines (`CINInit`, `CINLoad`, and so forth), see their descriptions in Chapter 1, *CIN Overview*.

The `.c` file created is a standard C file, except that LabVIEW gives the data types unambiguous names. C does not define the size of low-level data types—the `int` data type might correspond to a 16-bit integer for one compiler and a 32-bit integer for another compiler. The `.c` file uses names that are explicit about data type size, such as `int16`, `int32`, `float32`, and so on. LabVIEW comes with a header file, `extcode.h`, that contains typedefs associating these LabVIEW data types with the corresponding data type for the supported compilers of each platform.

`extcode.h` defines some constants and types whose definitions may conflict with the definitions of system header files. The LabVIEW `cintools` directory also contains a file, `hosttype.h`, that resolves these differences. This header file also includes many of the common header files for a given platform.

Note: ***You should always use*** `#include "extcode.h"` ***at the beginning of your source code. If your code needs to include system header files, you should include*** `"extcode.h"`, `"hosttype.h"`, ***and then any system header files, in that order.***

If you write a CIN that accepts a single 32-bit signed integer, the `.c` file indicates that the `CINRun` routine is passed an `int32` by reference. `extcode.h` typedefs an `int32` to the appropriate data type for the compiler you use (if it is a supported compiler); therefore, you can use the `int32` data type in external code that you write.

How LabVIEW Passes Fixed Sized Data to CINs

As described in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, you can designate terminals on the CIN as either input-output or output-only. Regardless of the designation, LabVIEW passes data by reference to the CIN. When modifying a parameter value, be careful to follow the rules described for each kind of terminal in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*. LabVIEW passes parameters to the `CINRun` routines in the same order as you wire data to the CIN—the first terminal pair corresponds to the first parameter, and the last terminal pair corresponds to the last parameter.

The following section describes how LabVIEW passes fixed sized parameters to CINs. See the *How LabVIEW Passes Variably Sized Data to CINs* section of this chapter for information on manipulating variably sized data such as arrays and strings.

Scalar Numerics

LabVIEW passes numeric data types to CINs by passing a pointer to the data as an argument. In C, this means that LabVIEW passes a pointer to the numeric data as an argument to the CIN. Arrays of numerics are described in the subsequent *Arrays and Strings* section of this chapter.

Scalar Booleans

LabVIEW stores Booleans in memory as 16-bit integers. If the high bit of the integer is 1, the Boolean is TRUE; otherwise the Boolean is FALSE. This is different from the more usual convention for Booleans, in which the low bit determines whether the Boolean is TRUE or FALSE. LabVIEW passes Booleans to CINs with the same

conventions as for numerics. LabVIEW stores arrays of Booleans differently; see the *Arrays and Strings* section of this chapter for more information.

Refnums

LabVIEW treats a refnum the same way as it treats a scalar number and passes refnums with the same conventions it uses for numbers.

Clusters of Scalars

For a cluster, LabVIEW passes a pointer to a structure containing the elements of the cluster. LabVIEW stores fixed-size values directly as components inside of the structure. If a component is another cluster, LabVIEW stores this cluster value as a component of the main cluster.

Return Value for CIN Routines

The names of the CIN routines are prefaced in the header file with the words `CIN MgErr`, as shown in the following example.

```
CIN MgErr CINRun(...);
```

The LabVIEW header file `extcode.h`, defines the word CIN to be either Pascal or nothing, depending on the platform. Prefacing a function with the word `Pascal` causes some C compilers to use Pascal calling conventions instead of C calling conventions to generate the code for the routine. LabVIEW uses Pascal calling conventions on the Macintosh when calling CIN routines, so the header file declares the CIN to be equivalent to Pascal on the Macintosh. On the PC and the Sun, however, LabVIEW uses standard C calling conventions, so the header file declares the CIN to be equivalent to nothing.

The `MgErr` data type is a LabVIEW data type that corresponds to a set of error codes that the manager routines return. If you call a manager routine that returns an error, you can either handle the error or return the error so that LabVIEW can handle it. If you can handle the errors that occur, return the error code `noErr`.

After calling a CIN routine, LabVIEW checks the `MgErr` value to determine whether an error occurred. If an error occurs, LabVIEW aborts the VI containing the CIN. If the VI is a subVI, LabVIEW aborts the VI that contains the subVI. This behavior enables LabVIEW to handle conditions when a VI runs out of memory. By aborting the

running VI, LabVIEW can possibly free enough memory to continue running correctly.

Examples with Scalars

The following examples show the steps for creating CINs and how to work with scalar data types. Chapter 5, *Manager Overview*, contains more examples.

Steps for Creating a CIN That Multiplies Two Numbers

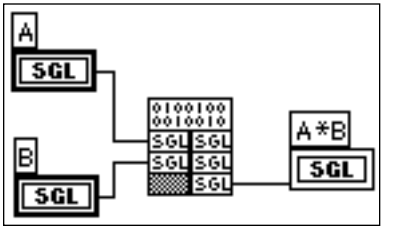
Consider a CIN that takes two single-precision floating-point numbers and returns their product.

1. Place the CIN on the Block Diagram

2. Add Two Input and Output Terminals to the CIN

3. Wire the Inputs and Outputs to the CIN

Place two single-precision numeric controls and one single-precision numeric indicator on a front panel. Wire the node as shown in the following illustration. Notice that **A*B** is wired to an output-only terminal pair.



Save the VI as `mult.vi`.

4. Create the CIN Source Code

Select **Create .c File...** from the CIN node pop-up menu. LabVIEW prompts you to select a name and a storage location for a `.c` file. Name the file `mult.c`. LabVIEW creates a `.c` file shown in the following listing.

```
/*
* CIN source file
*/

#include "extcode.h"

/* stubs for advanced CIN functions */

UseDefaultCINInit
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave

CIN MgErr CINRun (float32 *A, float32 *B,
float32 *A_B);

CIN MgErr CINRun (float32 *A, float32 *B,
float32 *A_B) {

  /* ENTER YOUR CODE HERE */

  return noErr;
  }
```

This `.c` file contains a prototype and a template for the CIN's `CINRun` routine. The `UseDefault...` macros shown in the preceding example code take the place of the corresponding CIN routines. LabVIEW calls the `CINRun` routine when the CIN executes. In this example, LabVIEW passes `CINRun` the addresses of the three 32-bit floating-point numbers. The parameters are listed left to right in the same order as you wired them (top to bottom) to the CIN. Thus, `A`, `B`, and `A_B` are pointers to **A**, **B**, and **A*B**, respectively.

As described in the *CIN .c File* section of this chapter, the `float32` data type is not a standard C data type. When LabVIEW creates a `.c` file, it gives unambiguous names for data types. For most C compilers, the `float32` data type corresponds to the `float` data type. However, this may not be true in all cases, because the C standard does not define the sizes for the various data types. You can use these LabVIEW data types in your code because `extcode.h` associates these data types

with the corresponding C data type for the compiler you are using. In addition to defining LabVIEW data types, `extcode.h` also prototypes LabVIEW routines that you can access. These data types and routines are described in Chapter 5, *Manager Overview*, of this manual and in the Online Reference.

☞ **Note:** ***The line*** `#include "extcode.h"` ***must be a full pathname to*** `extcode.h` ***under THINK C. For example:*** `#include "harddrive:cintools:extcode.h"`

Optionally, System 7.x users can use the `Aliases` ***folder technique described in the*** *THINK C for 68K* ***subsection of*** *Chapter 1, CIN Overview****, to enable the include line to read*** `#include "extcode.h"`**.**

For this multiplication example, simply fill in the code for the `CINRun` routine. You do not have to use the variable names that LabVIEW gives you in `CINRun`; you can change them to increase the readability of the code.

```
CIN MgErr CINRun (float32 *A, float32 *B,
float32 *A_B);
  {
  *A_B = *A**B;
  return noErr;
  }
```

`CINRun` multiplies the values to which `A` and `B` refer and stores the results in the location to which `A_B` refers. It is important that CIN routines return an error code, so that LabVIEW knows if the CIN encountered any fatal problems and handles the error correctly.

If you return a value other than `noErr`, LabVIEW stops the execution of the VI.

5. Compile the CIN Source Code

After creating the source code, you need to compile it and convert it into a form that LabVIEW can use. The following sections summarize the steps for each of the supported compilers.

☞ **Note:** ***Step 5 is different for each platform. Look under the heading for your platform and compiler in the following sections to find the instructions for your system. For details, refer to the relevant subsection within the*** *Compile the CIN Source Code* ***section in*** *Chapter 1, CIN Overview****.***

Macintosh

THINK C for 68K and Symantec C++

Create a new project and place `mult.c` in it. Build `mult.lsb` according to the instructions in the *THINK C for 68K* or the *Symantec C++* subsection of the *Compile the CIN Source Code* section of Chapter 1.

Macintosh Programmer's Workshop for 68K and Power Macintosh

Create a file named `mult.lvm`. Make sure the name variable is set to `mult`. Build `mult.lvm` according to the instructions in the *Macintosh Programmer's Workshop* subsection of the *Compile the CIN Source Code* section of Chapter 1.

Metrowerks CodeWarrior for Power Macintosh and 68K

Create a new project and place `mult.c` in it. Build `mult.lsb` according to the instructions in the *Metrowerks CodeWarrior* subsection of the *Compile the CIN Source Code* section of Chapter 1.

Microsoft Windows 3.*x*

Watcom C Compiler

Create a file named `mult.lvm`. Make sure the name variable is set to `mult`. Build `mult.lvm` according to the instructions in the *Watcom C Compiler* subsection of the *Compile the CIN Source Code* section of Chapter 1.

Microsoft Windows 95 and Windows NT

Microsoft SDK Compiler

Create a file named `mult.lvm`. Make sure the name variable is set to `mult`. Build `mult.lvm` according to the instructions in the *Microsoft SDK* subsection of the *Compile the CIN Source Code* section of Chapter 1.

Microsoft Visual C++ Compiler

Create a file named `mult.lvm`. Make sure the name variable is set to `mult`. Build `mult.lvm` according to the according to the instructions in the *Visual C++* subsection of the *Compile the CIN Source Code* section of Chapter 1. Add the following line to the top of the makefile.

```
IDE = VC
```

This line adds special VC++ libraries to the link. All other steps required to compile the CIN source code using the Visual C++ compiler are exactly the same as those for the Microsoft SDK compiler.

Solaris 1.*x*, Solaris 2.*x*, and HP-UX

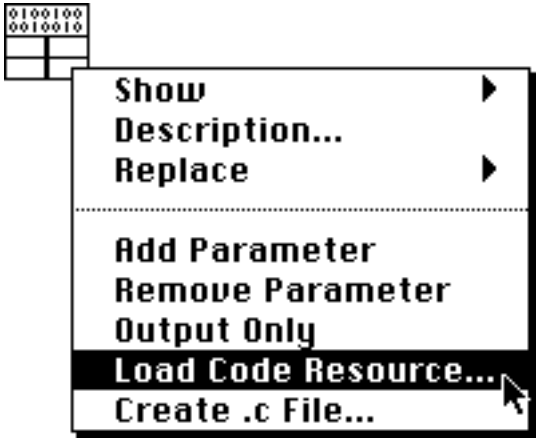
As described in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, you can create a makefile using the shell script `lvmkmf`. For this example, you should first enter the following command.

```
lvmkmf mult
```

This creates a file called `Makefile`. After executing `lvmkmf`, you should enter the standard `make` command, which uses `Makefile` to create a file called `mult.lsb`, which you can load into the CIN in LabVIEW.

6. Load the CIN Object Code

Now you are ready to load the CIN into LabVIEW and run it. Select **Load Code Resource** from the CIN pop-up menu and select `mult.lsb`, the object code file that you created.

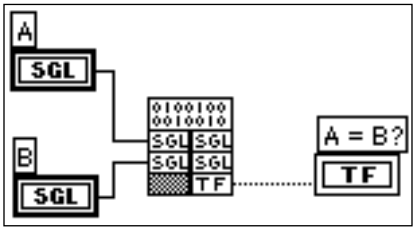


If you followed the preceding steps correctly, you should be able to run the VI at this point. If you save the VI, the CIN object code is saved along with the VI.

Comparing Two Numbers, Producing a Boolean Scalar

The following example shows how to create a CIN that compares two single-precision numbers. If the first number is greater than the second one, the return value is TRUE; otherwise, the return value is FALSE. This example gives only the block diagram and the code. Follow the instructions in the *Steps for Creating a CIN* section of Chapter 1 to create the CIN.

The diagram for this CIN is shown in the following illustration. Save the VI as `aequalb.vi`.



Create a `.c` file for the CIN, and name it `aequalb.c`. The `.c` file that LabVIEW creates is as follows.

```
/*
 * CIN source file
 */

#include "extcode.h"

/* stubs for advanced CIN functions */

UseDefaultCINInit
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave

CIN MgErr CINRun(float32 *ap, float32 *bp,
                    LVBoolean *aequalbp);
```

```
CIN MgErr CINRun(float32 *ap, float32 *bp,
                     LVBoolean *aequalbp) {
  if (*ap == *bp)
     *aequalbp= LVTRUE;
  else
     *aequalbp= LVFALSE;
  return noErr;
  }
```

How LabVIEW Passes Variably Sized Data to CINs

LabVIEW allocates memory for arrays and strings dynamically, and arrays and strings typically can grow quite large. If a string or array needs more space to hold new data, its current location may not offer enough contiguous space to hold the resulting string or array. In this case, LabVIEW may have to move the data to a location that offers more space.

To accommodate this relocation of memory, LabVIEW uses handles to refer to the storage location of variably sized data. A handle is a pointer to a pointer to the desired data. LabVIEW uses handles instead of simple pointers because handles allow LabVIEW to move the data without invalidating references from your code to the data. If LabVIEW moves the data, LabVIEW updates the intermediate pointer to reflect the new location. If you use the handle, references to the data go through the intermediate pointer, which always reflects the correct location of the data. Handles are described in detail in Chapter 5, *Manager Overview*, and information about specific handle functions is in the Online Reference.

Alignment Considerations

When a CIN returns variably sized data, you need to adjust the size of the handle that references the array. One method of adjusting the handle size is to use the memory manager routine `DSSetHandleSize` or, if the data is stored in the application zone, the `AZSetHandleSize` routine, to adjust the size of the return data handle. Both techniques work, but they are trouble-prone because you have to calculate the size of the new handle correctly. It is difficult to calculate the size correctly in a platform-independent manner, however, because some platforms have special requirements about how you align and pad memory.

Instead of using *XX*`SetHandleSize`, you should use the LabVIEW routines that take this alignment into account when resizing handles. You can use the `SetCINArraySize` routine to resize a string or an array of arbitrary data type. This function is described in the *Resizing Arrays and Strings* section of this chapter.

If you are not familiar with alignment differences for various platforms, the following examples highlight the problem. Keep in mind that `SetCINArraySize` and `NumericArrayResize` take care of these issues for you.

Consider the case of a 1D array of double-precision numbers. On the PC, an array of double-precision floating-point numbers is stored in a handle, and the first four bytes describe the number of elements in the array. These four bytes are followed by the 8-byte elements that make up the array. On the Sun, double-precision floating-point numbers must be aligned to 8-byte boundaries–the 4-byte value is followed by four bytes of *padding*. This padding ensures that the array data falls on eight-byte boundaries.

As a more complicated example, consider a three-dimensional array of clusters, in which each cluster contains a double-precision floating-point number and a 4-byte integer. As in the previous example, the Sun stores this array in a handle. The first 12 bytes contain the number of pages, rows, and columns in the array. These dimension fields are followed by four bytes of filler (which ensures that the first double-precision number is on an 8-byte boundary) and then the data. Each element contains eight bytes for the double-precision number, followed by four bytes for the integer. Each cluster is followed by four bytes of *padding*, which ensures that the next element is properly aligned.

Arrays and Strings

LabVIEW passes array by handle, as described in the *Alignment Considerations* section of this chapter. For an *n*-dimensional array, the handle begins with *n* 4-byte values that describe the number of values stored in a given dimension of the array. Thus, for a one-dimensional array, the first four bytes indicate the number of elements in the array. For a two-dimensional array, the first four bytes indicate the number of rows, and the second four bytes indicate the number of columns. These dimension fields can be followed by filler and then the actual data. Each element can also have padding to meet alignment requirements.

LabVIEW stores strings in memory as one-dimensional arrays of unsigned 8-bit integers.

LabVIEW stores Boolean arrays in memory as a series of bits packed to the nearest 16-bit word. LabVIEW ignores unused bits in the last word. LabVIEW orders the bits from left to right; that is, the most significant bit (MSB) is index 0. As with other arrays, a 4-byte dimension size precedes Boolean arrays. The dimension size for Boolean arrays indicates the number of Booleans contained in the array.

Paths (Path)

The exact structure for `Path` data types is subject to change in future versions of LabVIEW. A `Path` is a dynamic data structure that LabVIEW passes the same way it passes arrays. LabVIEW stores the data for `Path`s in an application zone handle. See the Online Reference, accessed from LabVIEW's Help window, or the *Code Interface Node Reference* online manual for descriptions of the functions that manipulate `Path`s.

Clusters Containing Variably Sized Data

For cluster arguments, LabVIEW passes a pointer to a structure containing the elements of the cluster. LabVIEW stores scalar values directly as components inside the structure. If a component is another cluster, LabVIEW stores this cluster value as a component of the main cluster. If a component is an array or string, LabVIEW stores a handle to the array or string component in the structure.

Resizing Arrays and Strings

You can use the LabVIEW `SetCINArraySize` routine to resize return arrays and strings that you pass to a CIN. You pass the function the handle that you want to resize, information that describes the data structure, and the desired size of the array or handle. The function takes into account any padding and alignment needed for the data structure. The function does not, however, update the dimension fields in the array. If you successfully resize the array, you need to update the dimension fields to correctly reflect the number of elements in the array.

You can resize numeric arrays more easily with `NumericArrayResize`. You pass to this function the array you

want to resize, a description of the data structure, and information about the new size of the array.

When you resize arrays of variably-sized data (for example, arrays of strings) with the `SetCINArraySize` or `NumericArrayResize` routines, you should be aware of the following facts. If the new size of the array is smaller, LabVIEW disposes of the handles used by the disposed element. Neither function sets the dimension field of the array. You must do this in your code after the function call. If the new size is larger, however, LabVIEW does not automatically create the handles for the new elements. You have to create these handles after the function returns.

The `SetCINArraySize` and `NumericArrayResize` functions are described in the following sections

SetCINArraySize

syntax `MgErr SetCINArraySize (UHandle dataH, int32 paramNum, int32 newNumElmts);`

`SetCINArraySize` resizes a data handle based on the data structure of an argument that you pass to the CIN. It does not set the array dimension field.

Parameter	Type	Description
dataH	`UHandle`	The handle that you want to resize.
paramNum	`int32`	The number for this parameter in the argument list to the CIN. The leftmost parameter has a parameter number of 0, and the rightmost has a parameter number of n-1, where n is the total number of parameters
newNumElmts	`int32`	The new number of elements to which the handle should refer. For a one-dimensional array of five values, you pass a value of 5 for this argument. For a two-dimensional array of two rows by three columns, you pass a value of 6 for this argument.

returns `MgErr`, which can contain the errors in the following list. `MgErr`s are discussed in Chapter 5, *Manager Overview*.

Error	Description

`noErr`
`mFullErr`
`mZoneErr`
	No error.
	Not enough memory to perform operation
	Handle is not in specified zone.

NumericArrayResize

syntax `MgErr NumericArrayResize(int32 typeCode, int32 numDims, UHandle *dataHP, int32 totalNewSize);`

`NumericArrayResize` resizes a data handle that refers to a numeric array. This routine also accounts for alignment issues. It does not set the array dimension field. If ***dataHP** is `NULL`, LabVIEW allocates a new array handle in ***dataHP**.

Parameter	Type	Description
typeCode	`int32`	Describes the data type for the array that you want to resize. The header file `extcode.h` defines the following constants for this argument `iB` Data is an array of signed 8-bit integers. `iW` is an array of signed 16-bit integers. `iL` Data is an array of signed 32-bit integers. `uB` Data is an array of unsigned 8-bit integers. `uW` Data is an array of unsigned 16-bit integers. `uL` Data is an array of unsigned 32-bit integers. `fS` Data is an array of single-precision (32-bit) numbers.

Parameter	Type	Description
		`fD` Data is an array of double-precision (64-bit) numbers. `fX` Data is an array of extended-precision numbers. `cS` Data is an array of single-precision complex numbers. `cD` Data is an array of double-precision complex numbers. `cX` Data is an array of extended-precision complex numbers.
numDims	`int32`	The number of dimensions in the data structure to which the handle refers. Thus, if the handle refers to a two-dimensional array, you pass a value of 2 for **numDims**.
***dataHP**	`UHandle`	A pointer to the handle that you want to resize. If this is a pointer to `NULL`, LabVIEW allocates and sizes a new handle appropriately and returns the handle in ***dataHP**.
totalNewSize	`int32`	The new number of elements to which the handle should refer. For a unidimensional array of five values, you pass a value of 5 for this argument. For a two-dimensional array of two rows by three columns, you pass a value of 6 for this argument.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mFullErr`	Not enough memory to perform operation
`mZoneErr`	Handle is not in specified zone.

Examples with Variably Sized Data

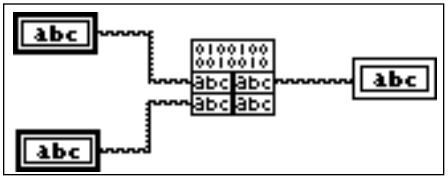
The following examples show the steps for creating CINs and how to work with variably-sized data types.

Concatenating Two Strings

The following example shows how to create a CIN that concatenates two strings. This example also shows how to use input-output terminals by passing the first string as an input-output parameter to the CIN. The top right terminal of the CIN returns the result of the concatenation.

This example gives only the diagram and the code. Follow the instructions in Chapter 1, *CIN Overview*, to create this CIN.

The diagram for this CIN is shown in the following illustration. Save the VI as `lstrcat.vi`.



Create a `.c` file for the CIN, and name it `lstrcat.h`. The `.c` file that LabVIEW creates is as follows.

```
/*
 * CIN source file
 */

#include "extcode.h"

/* stubs for advanced CIN functions */

UseDefaultCINInit
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave
```

```
CIN MgErr CINRun(
   LStrHandle var1,
   LStrHandle var2);

CIN MgErr CINRun(
   LStrHandle var1,
   LStrHandle var2) {

   /* ENTER YOUR CODE HERE */

   return noErr;
   }
```

Now fill in the CINRun function as follows:

```
CIN MgErr CINRun(
   LStrHandle strh1,
   LStrHandle strh2) {
   int32 size1, size2, newSize;
   MgErr err;

   size1 = LStrLen(*strh1);
   size2 = LStrLen(*strh2);
   newSize = size1 + size2;
   if(err = NumericArrayResize(uB, 1L,
      (UHandle*)&strh1, newSize))
        goto out;

   /* append the data from the second string to
      first string */
   MoveBlock(LStrBuf(*strh2),
              LStrBuf(*strh1)+size1, size2);

    /* update the dimension (length) of the
      first string */
   LStrLen(*strh1) = newSize;
out:
   return err;
   }
```

In this example, CINRun is the only routine that performs substantial operations. CINRun concatenates the contents of strh2 to the end of strh1, with the resulting string stored in strh1. Before performing the concatenation, you need to resize strh1 with the LabVIEW routine NumericArrayResize to hold the additional data.

If NumericArrayResize fails, it returns a non-zero value of type MgErr. In this case, NumericArrayResize could fail if LabVIEW does not have enough memory to resize the string. Returning the error code gives LabVIEW a chance to handle the error. If CINRun reports an error, LabVIEW aborts the calling VIs. Aborting the VIs may free up enough memory so that LabVIEW can continue running.

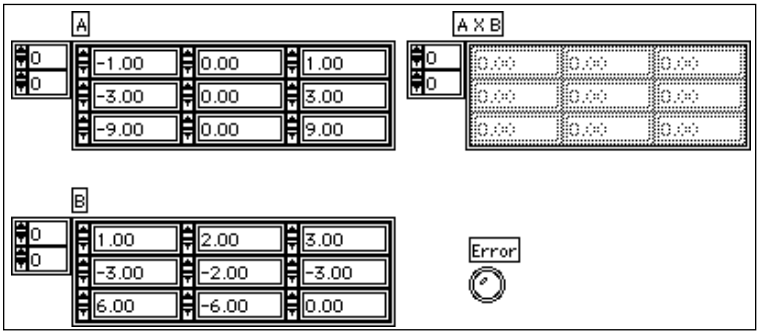
After resizing the string handle, this example copies the second string to the end of the first string using MoveBlock. MoveBlock is a support manager routine that moves blocks of data. Finally, this example sets the size of the first string to the length of the concatenated string.

Computing the Cross Product of Two Two-Dimensional Arrays

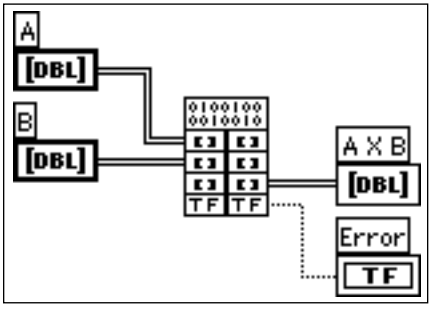
The following example shows how to create a CIN that accepts two two-dimensional arrays and then computes the cross product of the arrays. The CIN returns the cross product in a third parameter and a Boolean value as a fourth parameter. This Boolean is TRUE if the number of columns in the first matrix is not equal to the number of rows in the second matrix.

This example shows only the front panel, block diagram, and source code. Follow the instructions in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, to create the CIN.

The front panel for this VI is shown in the following illustration. Save the VI as `cross.vi`.



The block diagram for this VI is shown in the following illustration.



Save the `.c` file for the CIN as `cross.c`. Following is the source code for `cross.c` with the `CINRun` routine added.

```
/*
 * CIN source file
 */

#include "extcode.h"
#define ParamNumber 2
   /* The return parameter is parameter 2 */
#define NumDimensions 2
   /* 2D Array */
```

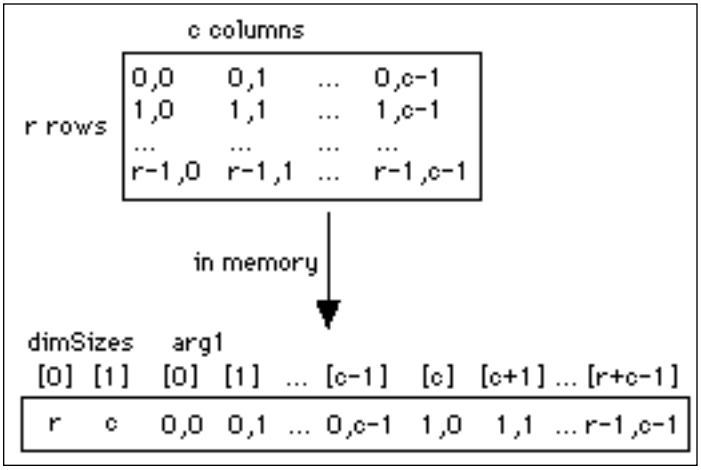
```
/* stubs for advanced CIN functions */

UseDefaultCINInit
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave

/*
 * typedefs
 */

typedef struct {
  int32 dimSizes[2];
  float64 arg1[1];
  } TD1;
typedef TD1 **TD1Hdl;

CIN MgErr CINRun(TD1Hdl ah, TD1Hdl bh, TD1Hdl
                    result, LVBoolean *errorp);
CIN MgErr CINRun(TD1Hdl ah, TD1Hdl bh, TD1Hdl
                    result, LVBoolean *errorp) {
  int32       i,j,k,l;
  int32       rows, cols;
  float64     *aElmtp, *bElmtp, *resultElmtp;
  MgErr       err=noErr;
  int32       newNumElmts;

  if ((k = (*ah)->dimSizes[1]) !=
                    (*bh)->dimSizes[0]) {
     *errorp = LVTRUE;
     goto out;
     }
  *errorp = LVFALSE;
  rows = (*ah)->dimSizes[0];
     /* number of rows in a and result */
  cols = (*bh)->dimSizes[1];
     /* number of cols in b and result */

  newNumElmts = rows * cols;
  if (err = SetCINArraySize((UHandle)result,
         ParamNumber, newNumElmts))
```

```
      goto out;

   (*resulth)->dimSizes[0] = rows;
   (*resulth)->dimSizes[1] = cols;

   aElmtp = (*ah)->arg1;
   bElmtp = (*bh)->arg1;
   resultElmtp = (*resulth)->arg1;

   for (i=0; i<rows; i++)
      for (j=0; j<cols; j++) {
          *resultElmtp = 0;
          for (l=0; l<k; l++)
              *resultElmtp += aElmtp[i*k + l] *
                          bElmtp[l*k + j];
          resultElmtp++;
          }
out:
   return err;
   }
```

In this example, `CINRun` is the only routine that performs substantial operations. `CINRun` cross multiplies the two-dimensional arrays `ah` and `bh`. LabVIEW stores the resulting array in `result`. If the number of columns in `ah` is not equal to the number of rows in `bh`, `CINRun` sets `*errorp` to `LVTRUE` to inform the calling diagram of invalid data.

`SetCINArraySize`, the LabVIEW routine that accounts for alignment and padding requirements, resizes the array. Notice that the two-dimensional array data structure is the same as the one-dimensional array data structure, except that the 2D array has two dimension fields instead of one. The two dimensions indicate the number of rows and the number of columns in the array, respectively.

The data is declared as a one-dimensional C-style array. LabVIEW stores data row by row, as shown in the following illustration.



For an array with `r` rows and `c` columns, you can access the element at row `i` and column `j` as shown in the following code fragment.

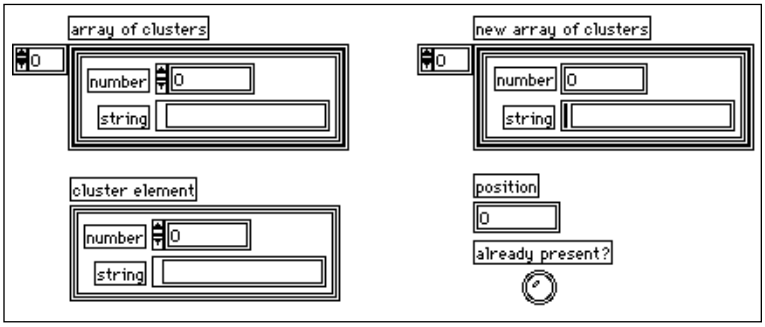
```
value = (*arrayh)->arg1[i*c + j];
```

Working with Clusters

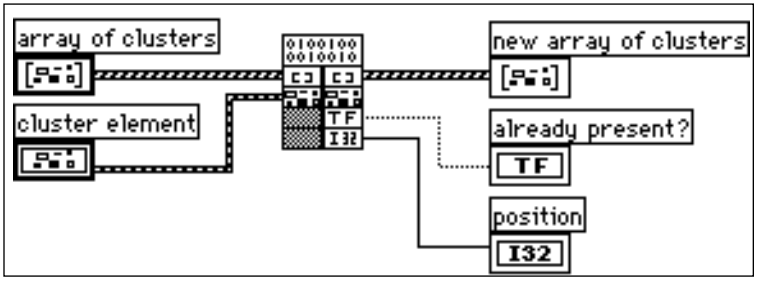
The following example takes an array of clusters and a single cluster as inputs, and the clusters contain a signed 16-bit integer and a string. The input for the array of clusters is an input-output terminal. In addition to the array of clusters, the CIN returns a Boolean and a signed 32-bit integer. If the cluster value is already present in the array of clusters, the CIN sets the Boolean to TRUE and returns the position of the cluster in the array of clusters using the 32-bit integer output. If the cluster value is not present, the CIN adds it to the array, sets the Boolean output to FALSE, and returns the new position of the cluster in the array of clusters.

This example shows only the front panel, block diagram, and source code. Follow the instructions in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, to create the CIN.

The front panel for this VI is shown in the following illustration. Save the VI as `tblsrch.vi`.



The block diagram for this VI is shown in the following illustration:



Save the `.c` file for the CIN as `tblsrch.c`. Following is the source code for `tblsrch.c` with the `CINRun` routine added:

```
/*
 * CIN source file
 */

#include "extcode.h"

#define ParamNumber 0
   /* The array parameter is parameter 0 */

/* stubs for advanced CIN functions */

UseDefaultCINInit
```

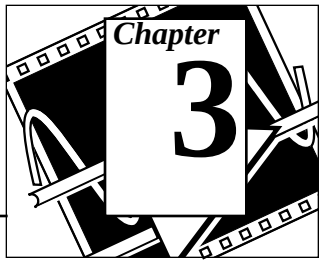
```
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave

/*
 * typedefs
 */

typedef struct {
  int16 number;
  LStrHandle string;
  } TD2;

typedef struct {
  int32 dimSize;
  TD2 arg1[1];
  } TD1;
typedef TD1 **TD1Hdl;

CIN MgErr CINRun(
                TD1Hdl      clusterTableh,
                TD2         *elementp,
                LVBoolean   *presentp,
                int32       *positionp);

CIN MgErr CINRun(
                TD1Hdl      clusterTableh,
                TD2         *elementp,
                LVBoolean   *presentp,
                int32       *positionp) {

  int32         size,i;
  MgErr         err=noErr;
  TD2           *tmpp;
  LStrHandle    newStringh;
  TD2           *newElementp;
  int32         newNumElements;

  size = (*clusterTableh)->dimSize;
  tmpp = (*clusterTableh)->arg1;
```

```
   *positionp = -1;
   *presentp = LVFALSE;

   for(i=0; i<size; i++) {
      if(tmpp->number == elementp->number)
         if(LStrCmp(*(tmpp->string),
                    *(elementp->string)) == 0)
            break;
      tmpp++;
      }

   if(i<size) {
      *positionp = i;
      *presentp = LVTRUE;
      goto out;
      }
   newStringh = elementp->string;
   if(err = DSHandToHand((UHandle *)
                         &newStringh))
      goto out;

   newNumElements = size+1;
   if(err =
      SetCINArraySize((UHandle)clusterTableh,
                      ParamNumber,
                      newNumElements)) {
      DSDisposeHandle(newStringh);
      goto out;
      }
   (*clusterTableh)->dimSize = size+1;

   newElementp = &((*clusterTableh)
                      ->arg1[size]);
   newElementp->number = elementp->number;
   newElementp->string = newStringh;

   *positionp = size;
out:
   return err;
   }
```

In this example, CINRun is the only routine that performs substantial operations. CINRun first searches through the table to see if the element is present. CINRun then compares string components using the LabVIEW routine LStrCmp, which is described in the Online Reference. If CINRun finds the element, the routine returns the position of the element in the array.

If the routine does not find the element, you have to add a new element to the array. Use the memory manager routine DSHandToHand to create a new handle containing the same string as the one in the cluster element that you passed to the CIN. CINRun then increases the size of the array using SetCINArraySize and fills the last position with a copy of the element that you passed to the CIN.

If the SetCINArraySize call fails, the CIN returns the error code returned by the manager. If the CIN is unable to resize the array, LabVIEW disposes of the duplicate string handle.



CIN Advanced Topics

This chapter covers several topics that are needed only in advanced applications, including how to use the `CINInit`, `CINDispose`, `CINAbort`, `CINLoad`, `CINUnload`, and `CINSave` routines. The chapter also discusses how global data works within CIN source code, and how users of Windows 3.1, Windows 95, and Windows NT can call a DLL from a CIN.

CIN Routines

A CIN consists of several routines, as described by the `.c` file that LabVIEW creates when you select **Create .c File...** from the CIN pop-up menu. The previous chapters have discussed only the `CINRun` routine. The other routines are `CINLoad`, `CINInit`, `CINAbort`, `CINSave`, `CINDispose`, and `CINUnload`.

It is important to understand that for most CINs, you need to write only the `CINRun` routine. The other routines are supplied mainly for the cases in which you have special initialization needs, such as when your CIN is going to maintain some information across calls, and you want to preallocate or initialize global state information.

In the case where you want to preallocate/initialize global state information, you first need to understand more of how LabVIEW manages data and CINs.

Data Spaces and Code Resources

When you create a CIN, you compile your source into an object code file and load the code into the node. At that point, LabVIEW loads a copy of the code (called a code resource) into memory and attaches it to the node. When you save the VI, this code resource is saved along with the VI as an attached component; the original object code file is no longer needed.

When LabVIEW loads a VI, it allocates a *data space*, a block of data storage memory, for that VI. This data space is used, for instance, to store the values in shift registers. If the VI is reentrant, then LabVIEW

allocates a data space for each usage of the VI. See Chapter 26, *Understanding How LabVIEW Executes VIs*, in your *LabVIEW User Manual* for more information on reentrancy.

Within your CIN code resource, you may have declared global data. Global data includes variables that are declared outside of the scope of all routines, and, for the purposes of this discussion, variables that are declared as static variables within routines. LabVIEW allocates space for this global data. As with the code itself, there is always only one instance of these globals in memory. Regardless of how many nodes reference the code resource and regardless of whether the surrounding VI is reentrant, there is only one copy of these globals in memory, and their values are consistent.

When you create a CIN node, LabVIEW allocates a *CIN data space*, a 4-byte storage location in the VI data space(s), strictly for the use of the CIN node. Each CIN may have one or more CIN data spaces reserved for the node, depending on how many times the node appears in a VI or collection of VIs. You can use this CIN data space to store global data on a per data space basis, as described in the *Code Globals and CIN Data Space Globals* section later in this chapter.

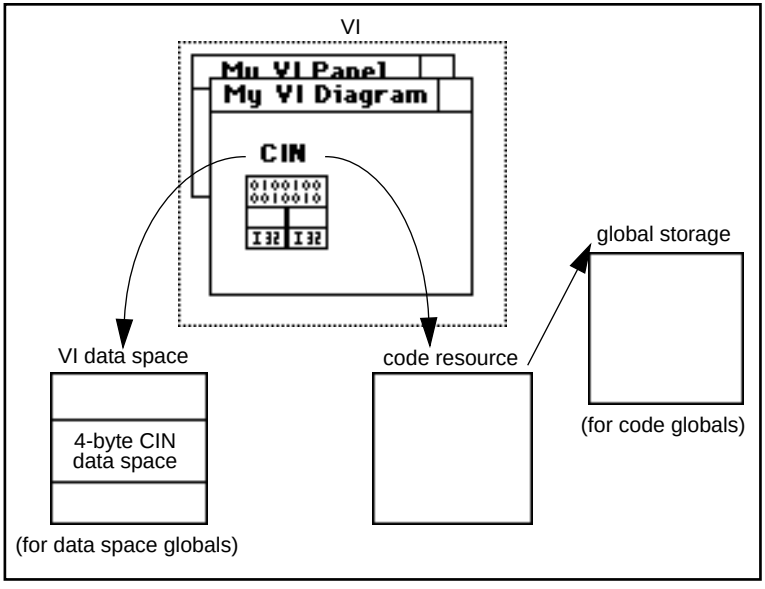


Figure 3-1. Data Storage Spaces for One CIN, Simple Case

A CIN node references the code resource by name, using the name you specified when you created the code resource. When you load a VI containing a CIN, LabVIEW looks in memory to see if a code resource with the desired name is already loaded. If so, LabVIEW links the CIN to the code resource for execution purposes.

This linking behaves the same way as links between VIs and subVIs. When you try to reference a subVI and another VI with the same name already exists in memory, LabVIEW references the one already in memory instead of the one you selected. In the same way, if you try to load references to two different code resources that have the same name, only one code resource is actually loaded into memory, and both references point to the same code. The difference is that LabVIEW can verify that a subVI call matches the subVI connector pane terminal, but LabVIEW cannot verify that your source code matches the CIN call.

CIN Routines: The Basic Case

The following discussion describes what happens in the standard case, in which you have a code resource that is referenced by only one CIN, and the VI that contains the CIN is non-reentrant. The other cases have slightly more complicated behavior, described in later sections of this chapter.

Loading a VI

When you first load a VI, LabVIEW calls the `CINLoad` routines for any CINs contained in that VI. This gives you a chance to load any file-based resources at load time, because LabVIEW calls this routine only when the VI is first loaded (see the *Loading a New Resource into the CIN* section that follows for an exception to this rule). After LabVIEW calls the `CINLoad` routine, it calls `CINInit`. Together, these two routines perform any initialization you need before the VI runs.

LabVIEW calls `CINLoad` once for a given code resource, regardless of the number of data spaces and the number of references to that code resource. This is why you should initialize code globals in `CINLoad`.

LabVIEW calls `CINInit` for a given code resource a total of one time for each CIN data space multiplied by the number of references to the code resource in the VI that corresponds to that data space. If you want to use CIN data space globals, you should initialize them in `CINInit`. See the *Code Globals and CIN Data Space Globals*, the *Loading a New*

Resource into the CIN, and the *Compiling a VI* sections of this chapter for related information.

Unloading a VI

When you close a VI front panel, LabVIEW checks to see if there are any references to that VI in memory. If so, then the VI code and data space remain in memory. When all references to a VI are removed from memory, and its front panel is not open, that VI is unloaded from memory.

When a VI is unloaded from memory, LabVIEW calls the `CINDispose` routine, giving you a chance to dispose of anything you allocated earlier. `CINDispose` is called for each `CINInit` call. For instance, if you used `XXNewHandle` in your `CINInit` routine, you should use `XXDisposeHandle` in your `CINDispose` routine. LabVIEW calls `CINDispose` for a code resource once for each individual CIN data space.

As the last reference to the code resource is removed from memory, LabVIEW calls the `CINUnload` routine for that code resource once, giving you the chance to dispose of anything that had been allocated in `CINLoad`. As with `CINDispose/CINInit`, a `CINUnload` is called for each `CINLoad`. For example, if you loaded some resources from a file in `CINLoad`, you can free the memory that those resources are using in `CINUnload`. After LabVIEW calls `CINUnload`, the code resource itself is unloaded from memory.

Loading a New Resource into the CIN

If you load a new code resource into a CIN, the old code resource is first given a chance to dispose of anything it needs to dispose. First, LabVIEW calls `CINDispose` for each CIN data space and each reference to the code resource, followed by the `CINUnload` for the old resource. The new code resource is then given a chance to perform any initialization that it needs to perform: LabVIEW calls the `CINLoad` for the new code resource, followed by the `CINInit` routine, called once for each data space and each reference to the code resource.

Compiling a VI

When you compile a VI, LabVIEW recreates the VI data space, resetting all uninitialized shift registers, for instance, to their default values. In the same way, your CIN is given a chance to dispose or

initialize any storage that it manages. Before disposing of the current data space, LabVIEW calls the `CINDispose` routine for each reference to the code resource within the VI(s) that are being compiled to give the code resource a chance to dispose of any old results it is managing. LabVIEW then compiles the VI and creates a new data space for the VI(s) being compiled (multiple data spaces for any VI that is reentrant). The `CINInit` routine is then called for each reference to the code resource within the VI(s) that were compiled to give the code resource a chance to create or initialize any data that it wants to manage.

Running a VI

When you press the Run button of a VI, that VI begins to execute. When LabVIEW encounters a code interface node, it calls the `CINRun` routine for that node.

Saving a VI

When you save a VI, LabVIEW calls the `CINSave` routine for that VI, giving you the chance to save any resources (for example, something you loaded in `CINLoad`). Notice that when you save a VI, LabVIEW creates a new version of the file, even if you are saving the VI with the same name. If the save is successful, LabVIEW deletes the old file and renames the new file with the original name. Therefore, anything you expect to be able to load in `CINLoad` needs to be saved in `CINSave`.

Aborting a VI

When you abort a VI, LabVIEW calls the `CINAbort` routine for every reference to a code resource contained in the VI that is being aborted. The `CINAbort` routine of all actively running subVIs is also called. If a CIN is in a reentrant VI, it is called for each CIN data space as well. CINs in VIs that are not currently executing are not notified by LabVIEW of the abort event.

CINs are synchronous. When a CIN begins execution, it takes over control of the program until the CIN completes. LabVIEW is not notified if the user clicks on the abort button and therefore cannot abort the CIN. No other LabVIEW tasks can execute while a CIN executes.

Multiple References to the Same CIN

If you have loaded the same code resource into multiple CINs, or you have duplicated a given code interface node, LabVIEW gives each reference to the code resource a chance to perform initialization or deallocation. No matter how many references you have in memory to a given code resource, the LabVIEW calls the `CINLoad` routine only once when the resource is first loaded into memory (though it is also called if you load a new version of the resource, as described in the previous section). When you unload the VI, LabVIEW calls `CINUnload` once.

After LabVIEW calls `CINLoad`, it calls `CINInit` once for each reference to the CIN, because its CIN data space may need initialization. Thus, if you have two nodes in the same VI, where both reference the same code, the LabVIEW calls the `CINLoad` routine once, and the `CINInit` twice. If you later load another VI that references the same code resource, then LabVIEW calls `CINInit` again for the new version. LabVIEW has already called `CINLoad` once, and does not call it again for this new reference.

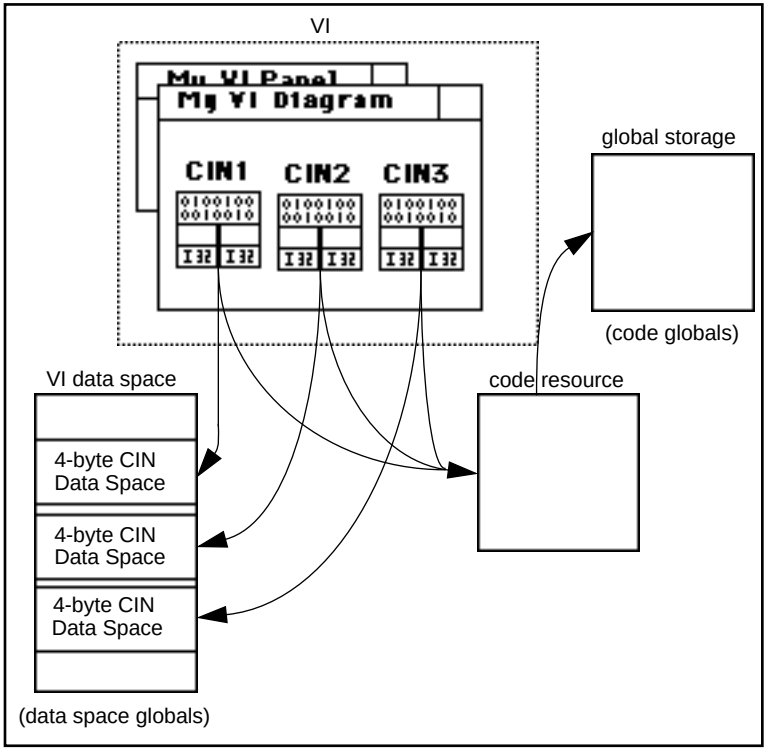


Figure 3-2. Three CINs Referencing the Same Code Resource

LabVIEW calls `CINDispose` and `CINAbort` for each individual CIN data space. LabVIEW calls `CINSave` only once, regardless of the number of references to a given code resource within the VI you are saving.

Reentrancy

When you make a VI reentrant, LabVIEW creates a separate data space for each usage of that VI. If you have a CIN data space in a reentrant VI and you call that VI in seven places, LabVIEW allocates memory to store seven CIN data spaces for that VI, each of which contains a unique storage location for the CIN data space for that calling instance.

As with multiple instances of the same node, LabVIEW calls the `CINInit`, `CINDispose`, and `CINAbort` routines for each individual CIN data space.

In the case where you have a reentrant VI that contains multiple copies of the same code resource, LabVIEW calls the `CINInit`, `CINDispose`, and `CINAbort` routines once for each use of the reentrant VI, multiplied by the number of references to the code resource within that VI.

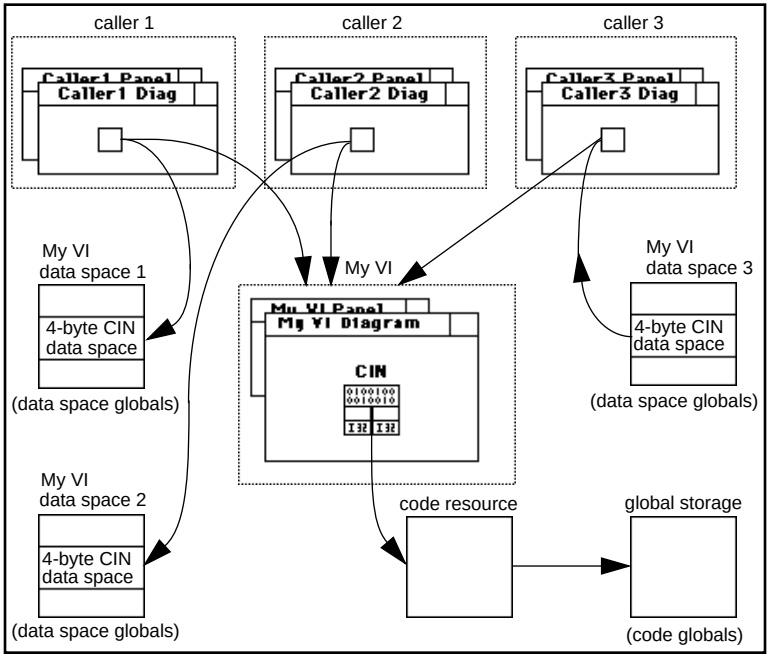


Figure 3-3. Three VIs Referencing a Reentrant VI Containing One CIN

Code Globals and CIN Data Space Globals

When you declare global or static local data within a CIN code resource, LabVIEW allocates storage for that data. LabVIEW maintains your globals across calls to various routines.

When you allocate a global in a CIN code resource, LabVIEW creates storage for only one instance of it, regardless of whether the CIN's VI is reentrant or whether you have multiple references to the same code resource in memory.

In some cases, you may want globals for each reference to the code resource multiplied by the number of usages of the VI (if the VI is reentrant). For each instance of one of these globals, LabVIEW allocates the CIN data space for the use of the code interface node. Within the `CINInit`, `CINDispose`, `CINAbort`, and `CINRun` routines you can call the `GetDSStorage` routine to retrieve the value of the CIN data space for the current instance. You can also call `SetDSStorage` to set the value of the CIN data space for this instance.

You can use this storage location to store any 4-byte quantity that you want to have for each instance of one of these globals. If you need more than four bytes of global data, you can store a handle or pointer to a structure containing your globals.

The following two lines of code are examples of the exact syntax of these two routines, defined in `extcode.h`.

```
int32 GetDSStorage(void);
```

This routine returns the value of the 4-byte quantity in the CIN data space that LabVIEW allocates for each CIN code resource, or for each use of the surrounding VI (if the VI is reentrant). You should call this routine only from `CINInit`, `CINDispose`, `CINAbort`, or `CINRun`.

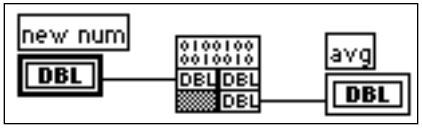
```
int32 SetDSStorage(int32 newVal);
```

This routine sets the value of the 4-byte quantity in the CIN data space that LabVIEW allocates for each CIN use of that code resource, or the uses of the surrounding VI, (if the VI is reentrant). It returns the old value of the 4-byte quantity in that CIN data space. You should call this routine only from `CINInit`, `CINDispose`, `CINAbort`, or `CINRun`.

Examples

The following two examples illustrate the differences between code globals and CIN data space globals. In both examples, the CIN takes a

number and returns the average of that number and the previous numbers passed to it.



When you design your code, decide whether it is appropriate to use code globals or data space globals. If you use code globals, calling the same code resource from multiple nodes or different reentrant VIs will affect the same set of globals. In the code globals averaging example, the result will indicate the average of all values passed to the CIN.

If you use CIN data space globals, each CIN that calls the same code resource and each VI, if the VI is reentrant, can have its own set of globals. In the CIN data space averaging example, the results would indicate the average of values passed to a specific node for a specific data space.

If you have only one CIN referencing the code resource, and the VI that contains that CIN is not reentrant, it does not matter which method you choose.

Using Code Globals

The following code implements averaging using code globals. Notice that the variables are initialized in `CINLoad`. If the variables are dynamically created (that is, if they are pointers or handles), you can allocate the memory for the pointer or handle in `CINLoad`, and deallocate it in `CINUnload`. You can do this because `CINLoad` and `CINUnload` are called only once, regardless of the number of references to the code resources and the number of data spaces. Notice that the `UseDefaultCINLoad` macro is not used, because this `.c` file has a `CINLoad` function.

```
/*
 * CIN source file
 */

#include "extcode.h"
```

```
float64 gTotal;
int32 gNumElements;

/* stubs for advanced CIN functions */

UseDefaultCINInit
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINUnload
UseDefaultCINSave

CIN MgErr CINRun(float64 *new_num, float64 *avg);

CIN MgErr CINRun(float64 *new_num, float64 *avg)
  {

  gTotal += *new_num;
  gNumElements++;
  *avg = gTotal / gNumElements;

  return noErr;
  }

CIN MgErr CINLoad(RsrcFile rf)
  {

  gTotal=0;
  gNumElements=0;

  return noErr;
  }
```

Using CIN Data Space Globals

The following is an alternative implementation of averaging using CIN data space globals. A handle for the global data is allocated in `CINInit`, and stored in the CIN data space storage using `SetDSStorage`. When LabVIEW calls the `CINInit`, `CINDispose`, `CINAbort`, or `CINRun` routines, it ensures that `GetDSStorage` and `SetDSStorage` will return the 4 byte CIN data space value for that node or CIN data space.

When you want to access that data, use `GetDSStorage` to retrieve the handle and then dereference the appropriate fields (see the code for

CINRun in the following example). Finally, in your CINDispose routine you need to dispose of the handle.

```
/*
 * CIN source file
 */

#include "extcode.h"

/* stubs for advanced CIN functions */

UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave

typedef struct {
   float64            total;
   int32              numElements;
   } dsGlobalStruct;

CIN MgErr CINInit() {
   dsGlobalStruct **dsGlobals;
   MgErr err = noErr;

   if (!(dsGlobals = (dsGlobalStruct **)
          DSNewHandle(sizeof(dsGlobalStruct))))
   {
   /* if 0, ran out of memory */
      err = mFullErr;
      goto out;
      }

   (*dsGlobals)->numElements=0;
   (*dsGlobals)->total=0;

   SetDSStorage((int32) dsGlobals);
out:
   return noErr;
   }

CIN MgErr CINDispose()
```

```
{
   dsGlobalStruct **dsGlobals;

   dsGlobals=(dsGlobalStruct **) GetDSStorage();

   if (dsGlobals)
      DSDisposeHandle(dsGlobals);

   return noErr;
   }

CIN MgErr CINRun(float64 *new_num, float64 *avg);

CIN MgErr CINRun(float64 *new_num, float64 *avg)
   {
   dsGlobalStruct **dsGlobals;

   dsGlobals=(dsGlobalStruct **) GetDSStorage();

   if (dsGlobals) {
      (*dsGlobals)->total += *new_num;
      (*dsGlobals)->numElements++;
      *avg = (*dsGlobals)->total /
                     (*dsGlobals)->numElements;

      }
   return noErr;
   }
```

Calling a Windows 95 or Windows NT Dynamic Link Library

No special techniques are necessary to call a Windows 95 or Windows NT DLL. Call DLLs the way you ordinarily would in a Windows 95 or Windows NT program.

Calling a Windows 3.1 Dynamic Link Library

Although dynamic link libraries (DLLs) can be called from a CIN, the method for doing so is somewhat cumbersome. The Call Library Function is a more convenient way to call a DLL, and the Watcom

compiler is not required. For more information on the Call Library Function, see Chapter 11, *Advanced Functions*, in the *LabVIEW Function Reference Manual*, and Chapter 24, *Calling Code from Other Languages*, in your *LabVIEW User Manual*.

Before attempting to link a dynamic link library with a CIN, you should first write a C program that calls it. The reasons for doing this are to ensure that you are calling the DLL properly, and that the DLL behaves as expected. You can test the C program using the debugging tools supplied by your compiler.

After you are sure that the DLL works and that you are calling it correctly, write the 32-bit CIN that LabVIEW can call. The main purpose of this CIN is to act as a go-between, translating LabVIEW 32-bit data to 16-bit data. This CIN will take 32-bit pointers from LabVIEW and then call the DLL with the appropriate arguments.

See the *Calling 16-bit DLLs* section of Chapter 37, *Programming Overview*, in the *Windows 32-bit Programming Guide* section of the *Watcom C/386 User's Guide* for a detailed discussion of how to call a 16-bit DLL.

No special techniques are necessary to call a Windows 95 or Windows NT DLL.

Calling a 16-Bit DLL

The following steps are a brief summary of how to call a 16-bit DLL from a CIN. If you are not familiar with the functions used in this example, you should refer to *Microsoft Windows Programmer's Reference* or the *Watcom C/386 User's Guide.*

1. Load the DLL

Load the DLL by calling the function `LoadLibrary()` with the name of the DLL. For example, the following code returns a handle to a specified library.

```
HANDLE hDLL;

hDLL = LoadLibrary("library name");
```

This is a standard Windows function, and is documented in the *Microsoft Windows Programmer's Reference*.

☞ **Note:** ***If you do not specify a full path, Windows searches the Windows directory, the Windows system directory, the LabVIEW directory, and the directories listed in the*** `Path` ***variable.***

2. Get the address of the desired function

Call `GetProcAddress()` with the name of the function that you want to call. For example, the following code returns the address of a specified function. This address is a 16-bit pointer, and cannot be called using standard DLL call methods. Instead you have to use the Watcom C method, shown as follows.

```
FARPROC lpfn;
lpfn = GetProcAddress(hDLL, "function name");
```

As with `LoadLibrary`, this function is a standard Windows function, and is documented in the *Microsoft Windows Programmer's Reference*.

3. Describe the function

Use `GetIndirectFunctionHandle()` to describe the function and the types of each parameter that it accepts. This function uses the following format.

```
HINDIR GetIndirectFunctionHandle(FARPROC proc
            [, long param1type,long param2type,
            ...,] long terminator);
```

proc is the address of the function that was returned in step 2.

The **paramXtype** values should be one of the following five constants that describe the parameters for the call to the function.

`INDIR_DWORD`	The parameter will be a long word value (a 32-bit integer).
`INDIR_WORD`	The parameter will be a word value (a 16-bit integer).
`INDIR_CHAR`	The parameter will be a byte value (an 8-bit integer).

INDIR_PTR	The parameter is a pointer. Watcom will automatically convert the 32-bit address to a 16-bit far pointer before calling the code. Notice that this 16-bit pointer is good only for the duration of the call; after the function returns, the 16-bit reference to the data is no longer valid.
INDIR_CDECL	Make the call using Microsoft C calling conventions. This keyword can be present anywhere in the parameter list.

For **terminator**, pass a value of INDIR_ENDLIST, which marks the end of the parameter list.

GetIndirectFunctionHandle() returns a handle that is used when you want to call the function.

4. Call the function

Use InvokeIndirectFunction() to call the function. Pass it the handle that was returned in step 3, along with the arguments that you want to pass to the CIN. This function uses the following format.

```
long InvokeIndirectFunction(HINDIR proc
            [, param1, param2, ...]);
```

proc is the address of the function returned in step 3. Following that are the parameters that you want to pass to the DLL.

Example: A CIN that Displays a Dialog Box

You cannot call most Windows functions directly from a CIN. You can, however, call a DLL, which in turn can call Windows functions. The following example shows how to call a DLL from a CIN. The DLL calls the Windows MessageBox function, which displays a window containing a specified message. This function returns after the user presses a button in the window.

The DLL

Most Windows compilers can create a DLL. Regardless of the compiler you use to create a DLL, the way you call it from a CIN will be roughly the same. Because you must have Watcom C/386 to write a Windows CIN, the following example is for a Watcom DLL. The process for creating a DLL using the Watcom compiler is described in Chapter 38, *Windows 32-Bit Dynamic Link Libraries*, of the *Watcom C/386 User's Guide*.

The following code is for a Watcom C/386 32-bit DLL that calls the `MessageBox` function. The `_16MessageBox` function calls the Windows `MessageBox` function; the only difference between these functions is that the former takes far 16-bit pointers, which are the type of pointers passed to the DLL. In this 32-bit environment, `MessageBox` expects near 32-bit pointers.

Passing pointers to 32-bit DLLs is inherently tricky. In this example, a 32-bit near pointer is converted to a 16-bit far pointer and passed to MessageBox via _16MessageBox. You cannot dereference a 16-bit pointer directly in this DL—it must first be converted to a 32-bit pointer. These pointer issues are not related to LabVIEW, but are unique to the Windows 3.1 environment. It may be helpful to build a rudimentary 32-bit Windows application (in place of LabVIEW) that calls the DLL to test the use of pointers.

The DLL function will accept two parameters. The first is the message to display in the window. The second is the title to display in the window. Both parameters are C strings, meaning that they are pointers to the characters of the string, followed by a terminating null character. Save the code in a file called `MSGBXDLL.C`.

```
/*
 * MSGBXDLL.C
 */
#include <windows.h>
#include <dos.h>
void FAR PASCAL Lib1( LPSTR message,
        LPSTR winTitle)
  {
  _16MessageBox( NULL,
        message,
        winTitle,
        MB_OK | MB_TASKMODAL );
```

```
  }
int PASCAL WinMain( HANDLE hInstance,
        HANDLE x1,
        LPSTR lpCmdLine,
        int x2 )
  {
  DefineDLLEntry( 1,
        (void *) Lib1,
        DLL_PTR,
        DLL_PTR,
        DLL_ENDLIST );
   return( 1 );
  }
```

In addition to the C file, you also need to create the following MSGBXDLL.LNK file.

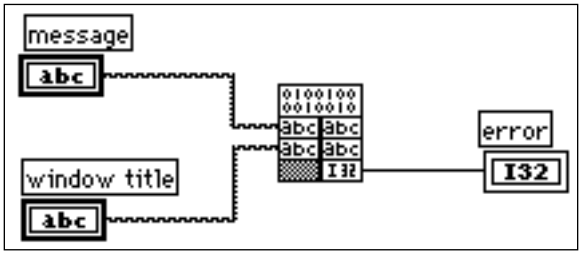
```
system win386
file msgbxdll
option map
option stack=12K
option maxdata=8K
option mindata=4K
```

Enter the following commands at the DOS prompt to create the DLL.

```
C>wcc386 msgbxdll /zw
C>wlink @msgbxdll
C>wbind msgbxdll -d -n
```

The Block Diagram

Following is the LabVIEW block diagram for a VI that calls a CIN that calls the previously described DLL. It passes two LabVIEW strings to the CIN, and the CIN returns an error code.



The CIN Code

The following C code is for a CIN that calls the DLL you created previously. This code assumes that the `.h` file created by LabVIEW is named `msgbox.h`.

This example does not pass a full path to `LoadLibrary`. You should move the DLL to the top level of your LabVIEW directory so that it will be found. See the note in the section *1. Load the DLL*, earlier in this chapter for more information.

```
/*
 * CIN source file
 */

#include "extcode.h"
#include "hosttype.h"
#include <windows.h>

/* stubs for advanced CIN functions */

UseDefaultCINInit
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave
```

```
CIN MgErr CINRun(LStrHandle message,
   LStrHandle winTitle,
   int32 *err)
   {
   HANDLE      hDLL = NULL;
   FARPROC     addr = NULL;
   HINDIR      hMessageBox;
   int         cb;
   char        *messageCStr = NULL,
               *winTitleCStr = NULL;
   MgErr       cinErr = noErr;

   *err=0;
   hDLL = LoadLibrary("msgbxdll.dll");
   if (hDLL < HINSTANCE_ERROR) {
      *err = 1;/* LoadLibrary failed */
      goto out;
      }

   addr = GetProcAddress(hDLL, "Win386LibEntry");
   if (!addr) {
      *err = 2;/* GetProcAddress failed */
      goto out;
      }

   hMessageBox = GetIndirectFunctionHandle(
               addr,
               INDIR_PTR,
               INDIR_PTR,
               INDIR_WORD,
               INDIR_ENDLIST );

   if (!hMessageBox) {
      *err = 3;/* GetIndirectFunctionHandle
                  failed */
      goto out;
      }

   if (!(messageCStr =
      DSNewPtr(LStrLen(*message)+1))) {
      /* mem errs are serious-stop execution */
              cinErr=mFullErr;
              goto out;
```

```
      }

   if (!(winTitleCStr =
      DSNewPtr(LStrLen(*winTitle)+1))) {
      /* mem errs are serious-stop execution */
            cinErr=mFullErr;
            goto out;
      }
   SPrintf(messageCStr, (CStr) "%P", *message);
   SPrintf(winTitleCStr, (CStr) "%P", *winTitle);

   cb = (WORD)InvokeIndirectFunction(
               hMessageBox,
               messageCStr,
               winTitleCStr,
               0x1 );

out:
   if (messageCStr)
      DSDisposePtr(messageCStr);
   if (winTitleCStr)
      DSDisposePtr(winTitleCStr);
   if (hDLL)
      FreeLibrary(hDLL);

   return cinErr;
   }
```

The CIN first loads the library, and then gets the address of the DLL entry point. As described in the *Watcom C/386 User's Guide*, a Watcom DLL has only one entry point, `Win386LibEntry`. Calling `GetProcAddress` for a Watcom DLL requests the address of this entry point. For a DLL created using a compiler other than the Watcom C compiler, request the address of the function you want to call.

To prepare for the DLL call after getting the address, the example calls `GetIndirectFunctionHandle`. Use this function to specify the data types for the parameters that you want to pass. The list is terminated with the `INDIR_ENDLIST` value. Because there is only one entry point with a Watcom DLL, pass an additional parameter (the `INDIR_WORD` parameter) that is the number of the routine you want to call in the DLL. With a DLL created using another compiler, you do

not need to pass a function number, because `GetProcAddress` returns the address of the desired function.

This example uses `InvokeIndirectFunction` to call the desired DLL function, passing the number of the routine that the example calls as the last parameter. With a DLL made by a compiler other than the Watcom C compiler, you don't need to pass the function number, because `GetProcAddress` returns the address of the desired function.

Notice that at each stage of calling the DLL, the code checks for errors and returns an error code if it fails.

Notice also that LabVIEW strings are different from C strings. C strings are terminated with a null character. LabVIEW strings are not null-terminated; instead, they begin with a four byte value that indicates the length of the string. Because the DLL expects C strings, this example creates temporary buffers for the C strings using `DSNewPtr`, and then uses `SPrintf` to copy the LabVIEW string into the temporary buffers. You might consider modifying the DLL to accept LabVIEW strings instead, because that would require no temporary copies of the strings.

Compiling the CIN

Following is the LabVIEW makefile for this CIN. It assumes that the `.c` file is named `msgbox.c`, the makefile is named `msgbox.lvm`, and the three pathnames for the directives `codeDir`, `cinToolsDir`, and `wcDir` are set correctly.

```
name=msgbox
type=CIN
codeDir=c:\labview\examples\cins\dll
cinToolsDir=c:\labview\cintools
wcDir=c:\wc
!include $(cinToolsDir)\generic.mak
```

The following command line prompt compiles the CIN.

```
c:> wmake /f msgbox.lvm
```

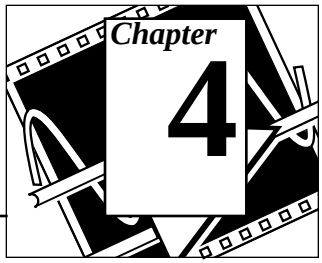
Optimization

To optimize the performance of this CIN call `LoadLibrary` during the `CINLoad` routine, and call `FreeLibrary` during the `CINUnload` routine. This keeps the overhead of loading and unloading the DLL from affecting your run-time performance. The following code shows the modifications you need to make to `CINRun`, `CINLoad`, and `CINUnload` to implement this optimization.

```
HANDLE hDLL = NULL;

CIN MgErr CINLoad(RsrcFile rf)
   {
   hDLL = LoadLibrary("msgbxdll.dll");
   return noErr;
   }

CIN MgErr CINRun(LStrHandle message,
   LStrHandle winTitle,
   int32 *err)
   {
   FARPROC    addr = NULL;
   HINDIR     hMessageBox;
   int        cb;
   char       *messageCStr = NULL,
              *winTitleCStr = NULL;
   MgErr      cinErr = noErr;

   *err=0;
   if (hDLL < HINSTANCE_ERROR) {
      *err = 1;/* LoadLibrary failed */
      goto out;
      }

   addr = GetProcAddress(hDLL,"Win386LibEntry");
   if (!addr) {
      *err = 2;/* GetProcAddress failed */
      goto out;
      }

   hMessageBox = GetIndirectFunctionHandle(
              addr,
              INDIR_PTR,
              INDIR_PTR,
```

```
                INDIR_WORD,
                INDIR_ENDLIST );

    if (!hMessageBox) {
       /* GetIndirectFunctionHandle failed */
       *err = 3;
       goto out;
       }

    if (!(messageCStr =
       DSNewPtr(LStrLen(*message)+1))) {
       /* mem errs are serious-stop execution */
       cinErr=mFullErr;
       goto out;
       }

    if (!(winTitleCStr =
       DSNewPtr(LStrLen(*winTitle)+1))) {
       /* mem errs are serious-stop execution */
       cinErr=mFullErr;
       goto out;
       }

    SPrintf(messageCStr, (CStr) "%P", *message);
    SPrintf(winTitleCStr, (CStr) "%P", *winTitle);

    cb = (WORD)InvokeIndirectFunction(
                hMessageBox,
                messageCStr,
                winTitleCStr,
                0x1 );

out:
    if (messageCStr)
       DSDisposePtr(messageCStr);
    if (winTitleCStr)
       DSDisposePtr(winTitleCStr);

    return cinErr;
    }

CIN MgErr CINUnload(void)
    {
```

```
if (hDLL)
   FreeLibrary(hDLL);

return noErr;
}
```



External Subroutines

This chapter describes how to create and call shared external subroutines from other external code modules.

Introduction

An external subroutine (or *shared* external subroutine) is a function that you can call from multiple external code modules. By placing common code in an external subroutine, you can avoid duplicating the code in each external code module. You can also use external subroutines to store information that must be accessible to multiple external code modules.

External subroutines are different from CINs in that LabVIEW diagrams do not call them directly. Instead, an external subroutine is a function that CINs and other external subroutines call. You store external subroutines in separate files, not in VIs.

When you load a VI that contains a CIN, LabVIEW determines whether the CIN references external subroutines. If it does, LabVIEW loads the external subroutines into memory and modifies the calling code so that it can call the subroutine. LabVIEW modifies any additional subroutines that reference the same external subroutine to reference the code already in memory. When you remove the last code that references the external subroutine from memory (when you close the VI containing the CIN), LabVIEW also unloads the external subroutine.

Placing code in external subroutines is helpful for several reasons.

- A single subroutine is easier to maintain, because you need update only a single file to affect all calls on the subroutine.
- A single subroutine can also reduce memory requirements, because only a single instance of the code is in memory, regardless of the number of calls to the subroutine.
- An external subroutine can maintain information used by multiple external code modules. The first time the external subroutine is

called, it can store data in a variable that is global to the external subroutine. Other external code modules can call the same external subroutine to retrieve the common data.

You store external subroutines as files, so you have to give each one a unique name. When LabVIEW searches for a subroutine file, it loads the first file it finds that has the correct name.

Note: ***External subroutines are not supported on the Power Macintosh. The Macintosh OS on the Power Macintosh provides a much cleaner mechanism for sharing code, namely, shared libraries. If you need to share code among multiple CINs on the Power Macintosh, consult your development environment documentation to learn how to build a shared library.***

Although external subroutines are supported on Solaris 2 and HP-UX, it is suggested that you use shared libraries instead.

Shared library mechanisms compatible with LabVIEW are available on all platforms. Under Microsoft Windows 3.1, Windows 95, and Windows NT, they are referred to as DLLs (dynamic link libraries). Under UNIX they are referred to as shared libraries or dynamic libraries.

Creating Shared External Subroutines

Normally, when you use a compiler to create a program, the compiler includes the code for all subroutines in a single file called the *executable*. External subroutines differ from standard subroutines in that you do not compile the code for the external subroutine with the code for the calling subroutine. Instead, your makefile, and consequently the code, indicate that the calling code references an external subroutine. LabVIEW loads external subroutines based on this information and links the calling code in memory, so that the calling code points correctly to the external subroutine.

You need to compile the calling code, even though its subroutines are not all present. LabVIEW must be able to determine that your code calls an external subroutine, find the subroutine, and load it into memory. When the subroutine is loaded, LabVIEW must be able to modify the memory image of the calling code so that it correctly references the memory location of the external code. Finally, LabVIEW may need to create and initialize memory space that the

external subroutine uses for global data. The following sections describe how to make this work.

External Subroutine

LabVIEW calls CINs, but only your code calls external subroutines. Instead of creating seven routines (`CINRun`, `CINSave`, and so on), you create only one entry point (`LVSBMain`) for an external subroutine. When another external code module calls this external subroutine, the `LVSBMain` subroutine executes.

`LVSBMain` is similar to `CINRun`. You can have an arbitrary number of parameters, and each parameter can be of arbitrary data type. Also, because only your code calls the subroutine, you can declare any return data type, and you do not need to place the word `CIN` in front of the function prototype. You will have to ensure that the parameters and return value are consistent between the calling and called code.

You compile an external subroutine almost the same way you compile a CIN. Because multiple external code modules can call the same external subroutine, LabVIEW does not load the code into a specific VI. Instead, LabVIEW loads the code from the file created by the makefile when the code is needed.

Macintosh

THINK C Compiler and CodeWarrior 68K Compiler

To make a subroutine using the THINK C Compiler, build the code resource (the `.tmp` file) as discussed in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, but leave out the `CINLib` library and select the **subroutine** option when running `lvsbutil.app`.

MPW Compiler

The only difference between the makefiles of subroutines and of CINs is that for a subroutine you specify a type of `LVSB` in your `.lvm` file instead of `CIN`. See the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, for a discussion of the makefile contents.

Microsoft Windows 3.1, Windows 95, and Windows NT

The only difference between the makefiles of subroutines and of CINs is that for a subroutine you specify a type of `LVSB` in your `.lvm` file

instead of CIN. See the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, for a discussion of the makefile contents.

Solaris 1.*x*, Solaris 2.*x*, and HP-UX

Unbundled Sun C Compiler and HP-UX C/ANSI C Compiler

The lvmkmf command for a CIN that calls an external subroutine is the same as described in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, except you use the -t option with the type LVSB to indicate that you are creating a LabVIEW subroutine instead of a CIN.

For example, if you want to create an external subroutine called find, you could use the following command:

```
lvmkmf -t LVSB find
```

This command creates a makefile that you could use to create the external subroutine.

Calling Code

You call external subroutines the same way that you call standard C subroutines. LabVIEW modifies the code at load time to ensure that the calling code passes control to the subroutine correctly.

When you call the external subroutine, do not use the function name LVSBMain to call the function. Instead, use the name you gave the external subroutine. If you created an external subroutine called fact.lsb, which in turn contained an LVSBMain() subroutine, for example, you should call the function as though it were named fact(). The argument list and return type should be the same as the argument and return type for the LVSBMain() subroutine.

You should also create a prototype for the function. This prototype should have the keyword extern so that the compiler will compile the CIN, even though the subroutine is not present.

When you create the makefile for the CIN, you identify the names of the external subroutines that the CIN calls. The LabVIEW makefile embeds information in your code that LabVIEW uses to determine that your code calls external subroutines. When you load external code that references external subroutines into a VI, LabVIEW searches for the

subroutine files. If it finds the subroutines, LabVIEW performs the appropriate linking. If a file is not found, LabVIEW displays a dialog box prompting you to find it. If you dismiss the dialog box without selecting the file, the VI loads into memory with a broken run arrow, indicating that the VI is not executable.

One way to ensure that LabVIEW can find external subroutines is to place them in the directories that you defined in the search path section of the LabVIEW defaults file. See the *Configuring LabVIEW* section of Chapter 8, *Customizing Your LabVIEW Environment*, of your *LabVIEW User Manual* for more information on setting path preferences.

Macintosh

THINK C Compiler

The THINK C project must have an extra file named `glue.c` that specifies each external subroutine. Each reference to the external subroutine should have an entry as follows in the `glue.c` file:

```
long gLVSB<external subroutine name> = 'LVSB';
void <external subroutine name>(void);
void <external subroutine name>(void) {
  asm {
     move.l gLVSB<external subroutine name>, a0
     jmp    (a0)
     }
  }
```

CodeWarrior 68K Compiler

The CodeWarrior project must have an extra file called glue.c, which specifies each external subroutine. Each reference to the external subroutine should have an entry as follows in the glue.c file:

```
long gLVSB<external subroutine name> = 'LVSB';
void <external subroutine name>(void);
asm void <external subroutine name>(void) {
     move.l gLVSB<external subroutine name>, a0
     jmp    (a0)
     }
```

MPW Compiler

The makefile for a calling CIN is the same as described in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, except you use the optional `subrNames` directive to identify the subroutines that the CIN references. Specifically, if your code calls two external subroutines, A and B, you need to have the following line in the makefile code:

```
subrNames = A B
```

Microsoft Windows 3.1, Windows 95, and Windows NT

The makefile for a calling CIN is the same as described in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, except you use the optional `subrNames` directive to identify the subroutines that the CIN references. Specifically, if your code calls two external subroutines, A and B, you need to have the following line in the code makefile, prior to the `!include` statement.

```
subrNames = A B
```

Solaris 1.*x*, Solaris 2.*x*, and HP-UX

Unbundled Sun C Compiler and HP-UX C/ANSI C Compiler

The `lvmkmf` command for a calling CIN is the same as described in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*, except you use the optional `-ext` option with the name of a file that lists the names of the subroutines called by the CIN, one name per line. The makefile that `lvmkmf` creates uses this file to append linkage information to the CIN object file.

For example, if your code calls two external subroutines, A and B, you create a new text file with the name A on the first line and B on the second. If the list of subroutines is in a file called `subrs`, and you want to call the calling CIN lookup, you can use the following command to create a makefile.

```
lvmkmf -ext subrs lookup
```

This command creates a makefile that you can use to create the CIN.

Simple Example

The following example illustrates the process of building an external subroutine that sums the elements of an array. This external subroutine can be used by a CIN that computes the mean and also by a CIN that computes the definite integral.

External Subroutine Example

As described in the *External Subroutine* section of this chapter, you must write a function called `LVSBMain()`. When you call the external subroutine from your CIN or another external subroutine, LabVIEW passes control to the `LVSBMain()` function. When you call the external subroutine, the arguments to it and to its return type should be the same as in the definition of `LVSBMain()`.

The following is the C code for this external subroutine. Name it `sum.c`.

```
/*
 * sum.c
 */
#include "extcode.h"
float64 LVSBMain(float64 *x, int32 n);
float64 LVSBMain(float64 *x, int32 n)
   {
   int32 i;
   float64 sum;

   sum = 0.0;
   for (i=0; i<n; i++)
      sum += *x++;

   return sum;
   }
```

Compiling the External Subroutine

Macintosh

THINK C Compiler and CodeWarrior 68K Compiler

To make a subroutine using the THINK C Compiler, create a project named `sum` and add `sum.c` and `LVSBLib` to the project. Do not include the `CINLib` file in your project. Set the options in the **Options...** and **Set Project Type** dialog boxes as described in the *Steps for Creating a CIN* section of Chapter 1, *CIN Overview*. After you create `sum.tmp`, run `lvsbutil.app` and select the **Subroutine** option.

MPW Compiler

As described in the *External Subroutine* section of this chapter, you compile an external subroutine the same way you compile a CIN. The first step is to create a makefile specification. Following are the contents of the makefile specification for this example. Notice that all `Dir` commands must end with a colon (`:`). Name the file `sum.lvm`.

`name = `*`name`*	`sum`
`type = `*`type`*	`LVSB`
`codeDir = `*`codeDir`*`:`	Complete pathname to the folder containing the `.c` file.
`cinToolsDir = `*`cinToolsDir`*`:`	Complete or partial pathname to the LabVIEW `cintools` folder.
`inclDir = `*`inclDir`*`:`	(optional) Complete or partial pathname to a folder containing any additional `.h` files.

Create the subroutine using the following command.

```
Directory <full pathname to CIN directory>
cinmake sum
```

Microsoft Windows 3.1

Watcom C Compiler

As described in the *External Subroutine* section of this chapter, you compile an external subroutine the same way you compile a CIN. The first step is to create a makefile specification. Following are the contents of the makefile specification for this example. Notice that all `Dir` commands must end *without* a backslash(\). Name the file `sum.lvm`.

`name = name` `sum`

`type = type` `LVSB`

`codeDir = codeDir` Complete pathname to the directory containing the `.c` file.

`cinToolsDir = cinToolsDir`
Complete or partial pathname to the LabVIEW `cintools` directory.

`inclDir = inclDir` (optional) Complete or partial pathname to a directory containing any additional `.h` files.

`wcDir = wcDir` Complete pathname to the directory containing Watcom.

```
!include $(cinToolsDir)\generic.mak
```

Create the subroutine using the following command.

```
wmake /f sum.lvm
```

Microsoft Windows 95 and Windows NT

As described in the *External Subroutine* section of this chapter, you compile an external subroutine the same way you compile a CIN. The first step is to create a makefile specification. Following are the contents of the makefile specification for this example. Name the file `sum.lvm`.

`name = name` `sum`

`type = type` `LVSB`

```
!include $(CINTOOLSDIR)\ntlvsb.mak
```

Create the subroutine using the following command.

```
nmake /f sum.lvm
```

Solaris 1.*x*, Solaris 2.*x*, and HP-UX

Unbundled Sun C Compiler and HP-UX C/ANSI C Compiler

As described in the *External Subroutine* section of this chapter, you compile an external subroutine the same way you compile a CIN. The first step is to create the makefile for the subroutine using the shell script `lvmkmf`. You can then use the standard **make** command to make the subroutine code. For this example, enter the following command.

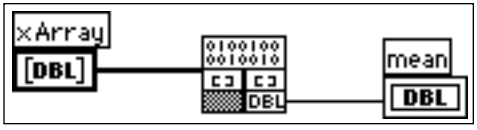
```
lvmkmf -t LVSB sum
```

This creates a file called `Makefile`. After executing `lvmkmf`, enter `make`, which uses the makefile to create a file called `sum.lsb`. CINs and other external subroutines can call this `sum.lsb` file.

Calling Code

The following example shows how to call an external subroutine. In this example, a CIN uses the external subroutine to calculate the mean of an array.

The diagram for the VI is shown in the following illustration. To avoid confusion, create the calling source code and makefiles in a directory separate from the external subroutine. Save the VI as `calcmean.vi`.



Save the `.c` file for the CIN as `calcmean.c`. The following is a listing of `calcmean.c`, with its `CINRun` routine filled in and the prototype for the `sum` external routine added.

```
/*
 * CIN source file
 */

#include "extcode.h"
```

```
/* stubs for advanced CIN functions */

UseDefaultCINInit
UseDefaultCINDispose
UseDefaultCINAbort
UseDefaultCINLoad
UseDefaultCINUnload
UseDefaultCINSave

/*
 * typedefs
 */

typedef struct {
  int32 dimSize;
  float64 arg1[1];
  } TD1;
typedef TD1 **TD1Hdl;

extern float64 sum(float64 *x, int32 n);

CIN MgErr CINRun(TD1Hdl xArray, float64 *mean);

CIN MgErr CINRun(TD1Hdl xArray, float64 *mean)
  {
  float64 *x, total;
  int32 n;

  x = (*xArray)->arg1;
  n = (*xArray)->dimSize;
  total = sum(x, n);
  *mean = total/(float64)n;
  return noErr;
  }
```

`CINRun` calculates the mean using the external subroutine `sum` to calculate the sum of the array. The external subroutine is declared with the keyword `extern` so that the code compiles even though the subroutine is not present.

Compiling the Calling Code

Macintosh

THINK C Compiler

The THINK C project must have an extra file called `glue.c` which specifies each external subroutine. The reference to the external subroutine `sum` should have an entry as follows in the `glue.c` file:

```
long gLVSBsum = 'LVSB';
void sum(void);
void sum(void) {
  asm {
     move.l gLVSBsum, a0
     jmp    (a0)
     }
  }
```

This is the entire text of the `glue.c` file.

CodeWarrior 68K Compiler

The CodeWarrior project must have an extra file called `glue.c`, which specifies each external subroutine. Each reference to the external subroutine `sum` should have an entry as follows in the `glue.c` file:

```
long gLVSBsum = 'LVSB';
void sum(void);
asm void sum(void){
     move.l gLVSBsum, a0
     jmp    (a0)
     }
```

This is the entire text of the `glue.c` file.

MPW Compiler

As described in the *Calling Code* section of this chapter, when you compile a CIN that references an external subroutine, you use the same makefile as described in the *Steps for Creating a CIN* section of the Chapter 1, *CIN Overview*, with the addition of a directive that identifies the subroutines that this CIN uses. Following are the contents of the

makefile specification for this example. Notice that the `Dir` command must end in a colon (`:`). Name the makefile `calcmean.lvm`.

`name = name`	`calcmean`
`type = type`	`CIN`
`codeDir = codeDir`:	Complete pathname to the folder containing the `.c` file.
`cinToolsDir = cinToolsDir`:	Complete or partial pathname to the LabVIEW `cintools` folder.
`inclDir = inclDir`:	(optional) Complete or partial pathname to a folder containing any additional `.h` files.
`subrNames = subrNames`	`sum`

Create the CIN using the following command.

```
Directory <full pathname to CIN directory>
cinmake sum
```

Microsoft Windows 3.1

Watcom C Compiler

As described in the *Calling Code* section of this chapter, when you compile a CIN that references an external subroutine, you use the same makefile as described in the *Steps for Creating a CIN* section of the Chapter 1, *CIN Overview*, with the addition of a directive that identifies the subroutines that this CIN uses. Following are the contents of the makefile specification for this example. Notice that the `Dir` command must end *without* a backslash (\). Name the makefile `calcmean.lvm`.

`name = name`	`calcmean`
`type = type`	`CIN`
`codeDir = codeDir`	Complete pathname to the directory containing the `.c` file.

`cinToolsDir = cinToolsDir`
Complete or partial pathname to the LabVIEW `cintools` directory.

`inclDir = inclDir`
(optional) Complete or partial pathname to a directory containing any additional `.h` files.

`wcDir = wcDir`
Complete pathname to the directory containing the Watcom C compiler.

```
subrNames = subrNames   sum
!include $(cinToolsDir)\generic.mak
```

Create the CIN using the following command.

```
wmake /f calcmean.lvm
```

Microsoft Windows 95 and Windows NT

As described in the *Calling Code* section of this chapter, when you compile a CIN that references an external subroutine, you use the same makefile as described in the *Steps for Creating a CIN* section of the Chapter 1, *CIN Overview*, with the addition of a directive that identifies the subroutines that this CIN uses. Following are the contents of the makefile specification for this example. Name the makefile `calcmean.lvm`.

```
name = name             calcmean
type = type             CIN
subrNames = subrNames   sum
!include $(CINTOOLSDIR)\ntlvsb.mak
```

Create the CIN using the following command.

```
nmake /f calcmean.lvm
```

Solaris 1.*x*, Solaris 2.*x*, and HP-UX

Unbundled Sun C Compiler and HP-UX C/ANSI C Compiler

As described in the *Calling Code* section of this chapter, when you compile a CIN that references an external subroutine, you use the

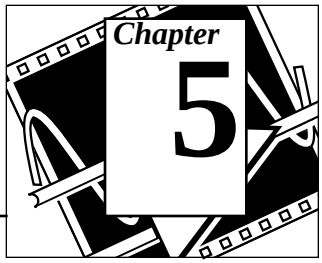
`lvmkmf` script with an addition directive that identifies a file with the names of all subroutines that the CIN calls.

For this example, create a text file with the name `meansubs`. It should contain a single line with the word `sum`.

You then create the makefile for this CIN using the following command.

```
lvmkmf -ext meansubs calcmean
```

This creates a file called `Makefile`. After executing `lvmkmf`, enter `make`, which uses the makefile to create a file called `calcmean.lsb`. You can load the `calcmean.lsb` file into the CIN.



Manager Overview

This chapter gives an overview of the function libraries, called *managers*, which you can use in external code modules. These include the memory manager, the file manager, and the support manager. The chapter also introduces many of the basic constants, data types, and globals contained in the LabVIEW libraries.

☞ **Note:** ***For descriptions of specific manager functions, see the*** *Function and VI Reference* ***topic in LabVIEW's Online Reference, or the*** *Code Interface Node Reference* ***online manual.***

Introduction

External code modules have a large set of functions you can use to perform simple and complex operations. These functions, organized into libraries called managers, range from low-level byte manipulation to routines for sorting data and managing memory. All manager routines described in this chapter are platform-independent. If you use these routines, you can create external code modules that will work on all platforms that LabVIEW supports.

A fundamental component of platform independence are data types that do not depend on the peculiarities of various compilers. The C language, for example, does not define the size of an integer. Without an explicit definition of the size of each data type, it is almost impossible to create code that works identically across multiple compilers.

LabVIEW managers use data types that explicitly indicate their size. For example, if a routine requires a 4-byte integer as a parameter, you define the parameter as an `int32`. The managers define data types in terms of the fundamental data types for each compiler. Thus, on one compiler, the managers might define an `int32` as an `int`, while on another compiler, the managers might define an `int32` as a `long int`. When writing external code modules, use the manager data types instead of the host computer data types, because your code will be more portable and have fewer errors.

Most applications need routines for allocating and deallocating memory on request. You can use the *memory manager* to dynamically allocate, manipulate, and release memory. The LabVIEW memory manager supports dynamic allocation of both non-relocatable and relocatable blocks, using pointers and handles. For more information, see the *Function and VI Reference* topic in LabVIEW's Online Reference, or the *Code Interface Node Reference* online manual.

Applications that manipulate files can use the functions in the *file manager*. This set of routines supports basic file operations such as creating, opening, and closing files, writing data to files, and reading data from files. In addition, file manager routines allow you to create directories, determine characteristics of files and directories, and copy files. File manager routines use a LabVIEW data type for file pathnames (`Path`s) that provides a platform-independent way of specifying a file or directory path. You can translate a `Path` to and from a host platform's conventional format for describing a file pathname. See the Online Reference for more information.

The *support manager* contains a collection of generally useful functions, such as functions for bit or byte manipulation of data, string manipulation, mathematical operations, sorting, searching, and determining the current time and date. See the Online Reference for more information.

Basic Data Types

Scalar Data Types

Note: ***The names of several types used by the manager routines have changed for version 4.0. These changes are strictly textual—that is, the actual types have not changed. The changes are:*** `boolean` ***was changed to*** `Bool32`***,*** `Ptr` ***was changed to*** `UPtr`***, and*** `Handle` ***was changed to*** `UHandle`***.***

Booleans

External code modules work with two kinds of Booleans—those that exist in LabVIEW block diagrams and those that pass to and from manager routines. The manager routines use a conventional form of Boolean, where 0 is FALSE and 1 is TRUE. This form of Boolean is called a `Bool32`, and it is stored as a 32-bit value.

LabVIEW block diagrams store Boolean scalars as 16-bit values. The high-bit is set if the Boolean is TRUE, and clear if the Boolean is FALSE. This form of Boolean is called an `LVBoolean`.

The two forms of Booleans are summarized in the following table.

Name	Description
`Bool32`	32-bit integer, 1 if TRUE, 0 if FALSE
`LVBoolean`	16-bit integer, high-bit set if TRUE, clear if FALSE

Numerics

The managers support 8-, 16-, and 32-bit signed and unsigned integers. For floating-point numbers, LabVIEW supports the single (32-bit), double (64-bit), and extended floating-point (at least 80-bit) data types. LabVIEW supports complex numbers that contain two floating-point numbers, with different complex numeric types for each of the floating-point data types. The following lists show the basic LabVIEW data types for numbers.

- Signed Integers
  - `int8` 8-bit integer
  - `int16` 16-bit integer
  - `int32` 32-bit integer
- Unsigned Integers
  - `uInt8` 8-bit unsigned integer
  - `uInt16` 16-bit unsigned integer
  - `uInt32` 32-bit unsigned integer
- Floating-Point Numbers
  - `float32` 32-bit floating-point number
  - `float64` 64-bit floating-point number
  - `floatExt` extended-precision floating-point number

In Windows, extended-precision numbers are stored as an 80-bit structure with two `int32` components, `mhi` and `mlo`, and an `int16` component, `e`. On the Sun, extended-precision numbers are stored as 128-bit floating-point numbers. On the 68K Macintosh, extended-precision numbers are stored in the 96-bit MC68881 format.

On the Power Macintosh, extended-precision numbers are stored in the 128-bit double-double format.

Complex Numbers

The complex data types are structures with two floating-point components, `re` and `im`. As with floating-point numbers, complex numbers can have 32-bit, 64-bit, and extended-precision components. The following segments of code give the type definitions for each of these complex data types.

```
typedef struct {
  float32 re, im;
  } cmplx64;
typedef struct {
  float64 re, im;
  } cmplx128;
typedef struct {
  floatExt re, im;
  } cmplxExt;
```

char Data Type

The `char` data type is defined by C to be a signed byte value. LabVIEW defines an unsigned `char` data type, with the following type definition.

```
typedef uInt8 uChar;
```

Dynamic Data Types

LabVIEW defines a number of data types that you must allocate and deallocate dynamically. *Arrays*, *strings*, and *paths* have data types that you must allocate using memory manager and file manager routines.

Arrays

LabVIEW supports arrays of any of the basic data types described in the *Scalar Data Types* section of this chapter. You can construct more complicated data types using clusters, which can in turn contain scalars, arrays, and other clusters.

The first four bytes of a LabVIEW array indicate the number of elements in the array. The elements of the array follow the length field. Chapter 2, *CIN Parameter Passing*, gives examples of how to manipulate arrays.

Strings

LabVIEW supports C-style strings and Pascal-style strings, a special string data type you use for string parameters to external code modules called `LStr`, and lists of strings. The support manager contains routines for manipulating strings and converting them among the different types of strings.

C-Style Strings (CStr)

A C string (`CStr`) is a series of zero or more unsigned characters, terminated by a zero. C strings have no effective length limit. Most manager routines use C strings, unless you specify otherwise. The following code segment is the type definition for a C-style string.

```
typedef uChar *CStr;
```

Pascal-Style Strings (PStr)

A Pascal string (`PStr`) is a series of unsigned characters. The value of the first character indicates the length of the string. This gives a range of 0 to 255 characters. The following code segment is the type definition for a Pascal-style string.

```
typedef uChar        Str255[256], Str31[32],
                     *StringPtr,
                     **StringHandle;
typedef uChar        *PStr;
```

LabVIEW Strings (LStr)

The first four bytes of a LabVIEW string (`LStr`) indicate the length of the string, and the specified number of characters follow. This is the string data type used by LabVIEW block diagrams. The following code segment is the type definition for an `LStr` string.

```
typedef struct {
  int32 cnt;
  /* number of bytes that follow */
  uChar str[1];
  /* cnt bytes */
  } LStr, *LStrPtr, **LStrHandle;
```

Concatenated Pascal String (CPStr)

Many algorithms require manipulation of lists of strings. Arrays of strings are usually the most convenient representation for lists. This representation can place a burden on the memory manager, however, because of the large number of dynamic objects that must be managed. To make working with lists more efficient, LabVIEW supports the concatenated Pascal string (`CPStr`) data type that is a list of Pascal-style strings concatenated into a single block of memory. You can use support manager routines to create and manipulate lists using this data structure.

This data type is defined as follows.

```
typedef struct {
  int32 cnt;
  /* number of pascal strings that follow */
  uChar str[1];
  /* cnt concatenated pascal strings */
  } CPStr, *CPStrPtr, **CPStrHandle;
```

Paths (Path)

A path (short for pathname) specifies the location of a file or directory in a computer's file system. There is a separate LabVIEW data type for a path (represented as `Path`), which the file manager defines in a platform-independent manner. The actual data type for a path is private to the file manager and subject to change. You create and manipulate `Path`s using file manager routines.

Memory-Related Types

LabVIEW uses pointers and handles to reference dynamically allocated memory. These data types are described in detail in the Online Reference and have the following type definitions.

```
typedef uChar *UPtr;
typedef uChar **UHandle;
```

Constants

The managers define the following constant for use with external code modules.

`NULL 0(uInt32)`

The following constants define the possible values of the `Bool32` data type.

```
FALSE 0 (int32)
TRUE 1 (int32)
```

The following constants define the possible values of the `LVBoolean` data type.

```
LVFALSE 0 (uInt16)
LVTRUE 0x8000 (uInt16)
```

Memory Manager

This section describes the memory manager, a set of platform-independent routines for allocating, manipulating, and deallocating memory from external code modules.

Read this section if you need to perform dynamic memory allocation or manipulation from external code modules. If your external code operates on data types other than scalars, you need to understand how LabVIEW manages memory and know the utilities that manipulate data.

Note: ***For descriptions of specific memory manager functions, see the*** *Function and VI Reference* ***topic in LabVIEW's Online Reference, or the*** *Code Interface Node Reference* ***online manual.***

Memory Allocation

Applications use two types of memory allocation: static and dynamic.

Static Memory Allocation

With static allocation, the compiler determines memory requirements when you create a program. When you launch the program, LabVIEW creates memory for the known global memory requirements of the application. This memory remains allocated while the program runs. This form of memory management is very simple to work with because the compiler handles all the details.

Static memory allocation cannot address the memory management requirements of most real-world applications, however, because you cannot determine most memory requirements until run-time. Also,

statically declared memory may result in larger memory requirements, because the memory is allocated for the life of the program.

Dynamic Memory Allocation: Pointers and Handles

With dynamic memory allocation, you reserve memory when you need it, and free memory when you are no longer using it. Dynamic allocation requires more work on your part than static memory allocation, because you have to determine memory requirements and allocate and deallocate memory as necessary.

The LabVIEW memory manager supports two kinds of dynamic memory allocation. The more conventional method uses pointers to allocate memory. With pointers, you request a block of memory of a given size, and the routine returns the address of the block to your CIN. When you no longer need the block of memory, you call a routine to free the block. You can use the block of memory to store data, and you reference that data using the address that the manager routine returned when you created the pointer. You can make copies of the pointer and use them in multiple places in your program to refer to the same data.

Pointers in the LabVIEW memory manager are nonrelocatable, which means that the manager never moves the memory block to which a pointer refers while that memory is allocated for a pointer. This avoids problems that occur when you need to change the amount of memory allocated to a pointer because other references would be out of date. If you need more memory, there might not be sufficient memory to expand the pointer's memory space without moving the memory block to a new location. This would cause problems if an application had multiple references to the pointer, because each pointer refers to the old memory address of the data. Using invalid pointers can cause severe problems.

A second form of memory allocation uses handles to address this problem. As with pointers, when you allocate memory using handles, you request a block of memory of a given size. The memory manager allocates the memory and adds the address of the memory block to a list of *master pointers*. The memory manager returns a handle that is a pointer to the master pointer. If you reallocate a handle and it moves to another address, the memory manager updates the master pointer to refer to the new address. As long as you look up the correct address using the handle, you access the correct data.

You use handles to perform most memory allocation in LabVIEW. Pointers are available, however, because in some cases they are more convenient and simpler to use.

Memory Zones

LabVIEW's memory manager interface has the ability to distinguish between two distinct sections, called zones. LabVIEW uses the data space (DS) zone only to hold VI execution data. LabVIEW uses the application zone (AZ) to hold all other data. Most memory manager functions have two corresponding routines, one for each of the two zones. Routines that operate on the data space zone begin with DS and routines for the application zone begin with AZ.

Currently, the two zones are actually one zone, but this may change in future releases of LabVIEW; therefore, a CIN programmer should write programs as if the two zones actually exist.

External code modules work almost exclusively with data created in the DS zone, although exceptions exist. In most cases, you use the DS routines when you need to work with dynamically allocated memory.

All data passed to or from a CIN is allocated in the DS zone except for `Path`s, which use AZ handles. You should only use file manager functions (not the AZ memory manager routines) to manipulate `Path`s. Thus, your CINs should use the DS memory routines when working with parameters passed from the block diagram. The only exceptions to this rule are handles created using the `SizeHandle` function, which allocates handles in the application zone. If you pass one of these handles to a CIN, your CIN should use AZ routines to work with the handle.

Using Pointers and Handles

Most memory manager functions have a DS routine and an AZ routine. In the following discussion, *`XXFunctionName`* refers to a function in a general context. In these situations, *`XX`* can be either *`DS`* or *`AZ`*. When a difference exists between the two zones, the specific function name is given.

You create a handle using `XXNewHandle`, with which you specify the size of the memory block. You create a pointer using `XXNewPtr`. `XXNewHandleClr` and `XXNewPtrClr` are variations that create the memory block and set it to all zeros.

When you are finished with the handle or pointer, release it using `XXDisposeHandle` or `XXDisposePtr`.

If you need to resize an existing handle, you can use the `XXSetHandleSize` routine. `XXGetHandleSize` determines the size of an existing handle. Because pointers are not relocatable, you cannot resize them.

A handle is a pointer to a pointer. In other words, a handle is the address of an address. The second pointer, or address, is a master pointer, which means that it is maintained by the memory manager. Languages that support pointers provide operators for accessing data by its address. With a handle, you use this operator twice; once to get to the master pointer, and a second time to get to the actual data. A simple example of how to work with pointers and handles in C is shown in the following section. Examples in the Online Reference show more complex ways to work with handles.

While operating within a single call of a CIN node, an AZ handle will not move unless you specifically resize it. In this context there is no need to lock or unlock handles. If your CIN maintains an AZ handle across different calls of the same CIN (an asynchronous CIN), the AZ handle may be relocated between calls. In this case, `AZHLock` and `AZHUnlock` may be useful if you do not want the handle to relocate. A DS handle will never move unless you resize it.

You can explicitly purge a handle using `XXEmptyHandle`. You can reallocate a purged master pointer using `XXReallocHandle`. Notice that `XXReallocHandle` does not actually recover the data to which the handle refers; this routine only reallocates a block of memory for the handle.

Additional routines make it easy to copy and concatenate handles and pointers to other handles, check the validity of handles and pointers, and copy or move blocks of memory from one place to another.

Simple Example

The following example code shows how you work with a pointer to an `int32`.

```
int32 *myInt32P;

myInt32P = (int32 *)DSNewPtr(sizeof(int32));
*myInt32P = 5;
```

```
x = *myInt32P + 7;
...

DSDisposePtr(myInt32P);
```

The first line declares the variable `myInt32P` as a pointer to, or the address of, a signed 32-bit integer. This does not actually allocate memory for the `int32`; it creates memory for an address and associates the name `myInt32P` with that address. The `P` at the end of the variable name is a convention used in this example to indicate that the variable is a pointer.

The second line creates a block of memory in the data space large enough to hold a single signed 32-bit integer and sets `myInt32P` to refer to this memory block.

The third line places the value 5 in the memory location to which `myInt32P` refers. The `*` operator refers to the value in the address location.

The fourth line sets `x` equal to the value at address `myInt32P` plus 7.

Finally, the last line frees the pointer.

The following code is the same example using handles instead of pointers.

```
int32 **myInt32H;

myInt32H =(int32**)DSNewHandle(sizeof(int32));
**myInt32H = 5;
x = **myInt32H + 7;
...
DSDisposeHandle(myInt32H);
```

The first line declares the variable `myInt32H` as a handle to an a signed 32-bit integer. Strictly speaking, this line declares `myInt32H` as a pointer to a pointer to an `int32`. As with the previous example, this declaration does not allocate memory for the `int32`; it creates memory for an address and associates the name `myInt32H` with that address. The `H` at the end of the variable name is a convention used in this example to indicate that the variable is a handle.

The second line creates a block of memory in the data space that is large enough to hold a single `int32`. `DSNewHandle` places the address of the memory block as an entry in the master pointer list and returns the address of the master pointer entry. Finally, this line sets `myInt32H` to refer to the master pointer.

The third line places the value 5 in the memory location to which `myInt32H` refers. Because `myInt32H` is a handle, you use the `*` operator twice to get to the data.

The fourth line sets `x` equal to the value referenced by `myInt32H` plus 7.

Finally, the last line frees the handle.

This example shows only the simplest aspects of how to work with pointers and handles in C. Other examples throughout this manual show different aspects of using pointers and handles. Refer to a C manual for a list of other operators that you can use with pointers and a more detailed discussion of how to work with pointers.

Reference to the Memory Manager

See the Online Reference for descriptions of the routines used for managing memory in external code segments of LabVIEW. For every function, if *XX* is `AZ`, the referenced handle, pointer, or block of memory is in the application zone. If *XX* is `DS`, the referenced handle, pointer, or block of memory is in the data space zone.

Memory Manager Data Structures

The memory manager defines generic handle and pointer data types as follows.

```
typedef uChar *Ptr;
typedef uChar **UHandle;
```

File Manager

This section describes the file manager, a set of platform-independent routines for creating and manipulating files and directories.

☞ **Note:** ***For descriptions of specific file manager functions, see the*** *Function and VI Reference* ***topic in LabVIEW's Online Reference, or the*** *Code Interface Node Reference* ***online manual.***

Introduction

The file manager supports routines for opening and creating files, reading data from and writing data to files, and closing files. In addition, with these routines you can manipulate the end-of-file mark of a file and position the current read or write mark to an arbitrary position in the file. With other file routines you can move, copy, and rename files, determine and set file characteristics and delete files.

The file manager contains a number of routines for directories. With these routines you can create and delete directories. You can also determine and set directory characteristics and obtain a list of a directory's contents.

LabVIEW supports concurrent access to the same file, so you can have a file open for both reading and writing simultaneously. When you open a file, you can indicate whether you want the file to be read from and written to concurrently. You can also lock a range of the file, if you need to ensure that a range is nonvolatile at a given time.

Finally, the file manager provides many routines for manipulating paths (short for pathnames) in a platform-independent manner. The file manager supports the creation of path descriptions, which are either relative to a specific location or absolute (the full path). With file manager routines you can create and compare paths, determine characteristics of paths, and convert a path between platform-specific descriptions and the platform-independent form.

Identifying Files and Directories

When you perform operations on files and directories, you need to identify the target of the operation. The platforms that LabVIEW supports use a hierarchical file system, meaning that files are stored in directories, possibly nested several levels deep. These file systems support the connection of multiple discrete storage media, called volumes. For example, DOS-based systems support multiple drives connected to the system. For most of these file systems, you must include the volume name to completely specify the location of a file. On other systems, such as UNIX, you do not specify the volume name

because the physical implementation of the file system is hidden from the user.

How you identify a target depends upon whether the target is an open or closed file. If a target is a closed file or a directory, you specify the target using the target's *path*. The path describes the volume containing the target, the directories between the top level and the target, and the target's name. If the target is an open file, you use a file descriptor to specify that LabVIEW should perform an operation on the open file. The file descriptor is an identifier that the file manager associates with the file when you open it. When you close the file, the file manager dissociates the file descriptor from the file.

Path Specifications

Conventional Path Specifications

All platforms have a method for describing the paths for files and directories. These path specifications are similar, but they are usually incompatible from one platform to another. You usually specify a path as a series of names separated by separator characters. Typically, the first name is the top level of the hierarchical specification of the path, and the last name is the file or directory that the path identifies.

There are two types of paths—*relative paths* and *absolute paths*. A relative path describes the location of a file or directory relative to an arbitrary location in the file system. An absolute path describes the location of a file or directory starting from the top level of the file system.

A path does not necessarily go from the top of the hierarchy down to the target. You can often use a platform-specific tag in place of a name that indicates that the path should go up a level from the current location.

For instance, on a UNIX system, you specify the path of a file or directory as a series of names separated by the slash (/) character. If the path is an absolute path, you begin the specification with a slash. You can indicate that the path should move up a level using two periods in a row (..). Thus, the following path specifies a file `README` relative to the top level of the file system.

```
/usr/home/gregg/myapps/README
```

Two relative paths to the same file are as follows.

`gregg/myapps/README`	relative to `/usr/`home
`../myapps/README`	relative to a directory inside of the `gregg` directory

On the PC, you separate names in a path with a backslash (\ ) character. If the path is an absolute path, you begin the specification with a drive designation, followed by a colon (:), followed by the backslash. You can indicate that the path should move up a level using two periods in a row (..). Thus, the following path specifies a file `README` relative to the top level of the file system, on a drive named `C`.

```
C:\HOME\GREGG\MYAPPS\README
```

Two relative paths to the same file are as follows.

`GREGG\MYAPPS\README`	relative to the `HOME` directory
`..\MYAPPS\README`	relative to a directory inside of the `GREGG` directory

On the Macintosh, you separate names in a path with the colon (:) character. If the path is an absolute path, you begin the specification with the name of the volume containing the file. If an absolute path consists of only one name (it specifies a volume), it must end with a colon. If the path is a relative path, it begins with a colon. This colon is optional for a relative path consisting of only one name. You can indicate that the path should move up a level using two colons in a row (::). Thus, the following path specifies a file `README` relative to the top level of the file system, on a drive named `Hard Drive`.

```
Hard Drive:Home:Gregg:MyApps:README
```

Two relative paths to the same file are as follows.

`:Gregg:MyApps:README`	relative to the `Home` directory
`::MyApps:README`	relative to a directory inside of the `Gregg` directory

Empty Path Specifications

In LabVIEW you can define a path with no names, called an *empty path*. An empty path is either absolute or relative. The empty absolute path is the highest point you can specify in the file hierarchy. The empty relative path is a path relative to an arbitrary location in the file system to itself.

On a UNIX system, a slash (/) represents the empty absolute path. The slash specifies the root of the file hierarchy. A period (.) represents the empty relative path.

On the PC, you represent the empty absolute path as an empty string. It specifies the set of all volumes on the system. A period (.) represents the empty relative path.

On the Macintosh, the empty absolute path is represented as an empty string. It specifies the set of all volumes on the system. A colon (:) represents the empty relative path.

LabVIEW Path Specification

In LabVIEW, you specify a path using a special LabVIEW data type, represented as `Path`. The exact structure of the `Path` data type is private to the file manager. You create and manipulate the `Path` data type using file manager routines.

A `Path` is a dynamic data structure. Just as you use memory manager routines to allocate and deallocate handles and pointers, you use file manager routines to create and deallocate `Path`s. Just as with handles, declaring a `Path` variable does not actually create a `Path`. Before you can use the `Path` to manipulate a file, you must dynamically allocate the `Path` using file manager routines. When you are finished using the `Path` variable, you should release the `Path` using file manager routines.

In addition to providing routines for the creation and elimination of `Path`s, the file manager provides routines for comparing `Path`s, duplicating `Path`s, determining characteristics of `Path`s, and converting `Path`s to and from other formats, such as the platform-specific format for the system on which LabVIEW is running.

File Descriptors

When you open a file, LabVIEW returns a file descriptor associated with the file. A file descriptor is a data type LabVIEW uses to identify open files. All operations performed on an open file use the file descriptor to identify the file.

A file descriptor is valid only while the file is open. If you close the file, the file descriptor is no longer associated with the file. If you subsequently open the file, the new file descriptor will most likely be different from the file descriptor LabVIEW used previously.

File Refnums

In the file manager, LabVIEW accesses open files using file descriptors. On the front panel and block diagram, however, LabVIEW accesses open files through file refnums. A file refnum contains a file descriptor for use by the file manager, and additional information used by LabVIEW.

LabVIEW specifies file refnums using the `LVRefNum` data type, the exact structure of which is private to the file manager. If you want to pass references to open files into or out of a CIN, use the functions in the *Manipulating File Refnums* topic of the Online Reference to convert file refnums to file descriptors, and to convert file descriptors to file refnums.

Support Manager

The support manager is a collection of constants, macros, basic data types, and functions that perform operations on strings and numbers. The support manager also has functions for determining the current time in a variety of formats.

Note: ***This section gives only a brief overview of the support manager. For descriptions of specific support manager functions, see the*** *Function and VI Reference* ***topic in LabVIEW's Online Reference, or the*** *Code Interface Node Reference* ***online manual.***

The support manager's string functions contain much of the functionality of the string libraries supplied with standard C compilers, such as string concatenation and formatting. You can use variations of many of these functions with LabVIEW strings (4-byte length field followed by data, generally stored in a handle), Pascal strings (1-byte length field followed by data), and C strings (data terminated by a null character).

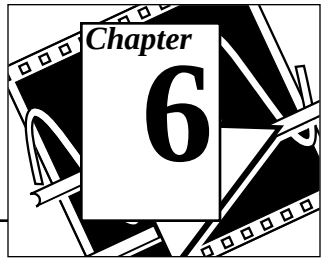
With the utility functions you can sort and search on arbitrary data types, using quicksort and binary search algorithms.

The support manager contains transcendental functions for many complex and extended floating-point operations.

Certain routines specify time as a data structure with the following form.

```
typedef struct {
   int32 sec;/* 0:59 */
   int32 min;/* 0:59 */
   int32 hour;/* 0:23 */
   int32 mday;/* day of the month, 1:31 */
   int32 mon;/* month of the year, 1:12 */
   int32 year;/* year, 1904:2040 */
   int32 wday;/* day of the week, 1:7 for Sun:Sat */
   int32 yday;/* day of year (julian date), 1:366 */
   int32 isdst;/* 1 if daylight savings time */
   } DateRec;
```

Memory Manager Functions



Allocating and Releasing Handles

AZDisposeHandle
DSDisposeHandle

syntax

```
MgErr       AZDisposeHandle(h);
MgErr       DSDisposeHandle(h);
```

`XXDisposeHandle` releases the memory referenced by the specified handle.

Parameter	Type	Description
h	`UHandle`	Handle you want to dispose of.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mZoneErr`	Handle or pointer not in specified zone.

AZEmptyHandle
DSEmptyHandle

syntax

```
MgErr       AZEmptyHandle(h);
MgErr       DSEmptyHandle(h);
```

`XXEmptyHandle` releases the memory referenced by a handle, and replaces the handle's master pointer with `NULL`.

The master pointer is set to `NULL`, but remains a valid master pointer after this call. All handle-based references to the block of memory point to the `NULL` handle. If you reallocate space for the handle using `XXReallocHandle`, all references to the old handle will reference the new block of memory.

Parameter	Type	Description
h	`UHandle`	Handle to empty.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mZoneErr`	Handle or pointer not in specified zone.

AZGetHandleSize
DSGetHandleSize

syntax

```
int32       AZGetHandleSize(h);
int32       DSGetHandleSize(h);
```

`XXGetHandleSize` returns the size of the block of memory referenced by the specified handle.

Parameter	Type	Description
h	`UHandle`	Handle whose size you want to determine.

returns The size in bytes of the relocatable block referenced by the handle **h**. If an error occurs, `XXGetHandleSize` returns a negative number.

AZNewHandle
DSNewHandle

syntax

```
UHandle     ZNewHandle(size);
UHandle     DSNewHandle(size);
```

`XXNewHandle` creates a new handle to a relocatable block of memory of the specified **size**. The routine aligns all handles and pointers in DS to accommodate the largest possible data representations for the platform in use.

Parameter	Type	Description
size	`int32`	Size, in bytes, of the handle to create.

returns A handle of the specified size. Returns `NULL` if the routine fails.

AZNewHClr
DSNewHClr

syntax

```
UHandle     AZNewHClr(size);
UHandle     DSNewHClr(size);
```

`XXNewHClr` creates a new handle to a relocatable block of memory of the specified **size** and initializes the memory to zero.

Parameter	Type	Description
size	`int32`	Size, in bytes, of the handle to create.

returns A handle of the specified size, where the block of memory is set to all zeros. Returns `NULL` if the routine fails.

AZReallocHandle
DSReallocHandle

syntax

```
MgErr       AZReallocHandle(h, size);
MgErr       DSReallocHandle(h, size);
```

`XXReallocHandle` creates a new block of memory and sets the specified handle to reference the block of memory.

If **h** is not already an empty handle, the function releases the block of memory referenced by **h** before creating the new block. A handle is an empty handle if you called `XXEmptyHandle` on the handle, or if you marked the handle as purgeable and the memory manager purged it from memory.

Parameter	Type	Description
h	`UHandle`	Handle to recover.
size	`int32`	New size, in bytes, of the handle.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mgArgErr`	Invalid argument.
`mFullErr`	Not enough memory to perform operation.
`mZoneErr`	Handle or pointer not in specified zone.

AZRecoverHandle
DSRecoverHandle

syntax

```
UHandle     AZRecoverHandle(p);
UHandle     DSRecoverHandle(p);
```

Given a pointer to a block of memory that was originally declared as a handle, `XXRecoverHandle` returns a handle to the block of memory.

This function is useful when you have the address of a block of memory that you know is a handle, and you need to get a true handle to the block of memory.

Parameter	Type	Description
p	`UPtr`	Pointer to a relocatable block of memory.

returns A handle to the block of memory to which **p** refers. Returns `NULL` if the routine fails.

AZSetHandleSize
DSSetHandleSize

syntax

```
MgErr       AZSetHandleSize(h, size);
MgErr       DSSetHandleSize(h, size);
```

`XXSetHandleSize` changes the size of the block of memory referenced by the specified handle.

While LabVIEW arrays are stored in DS handles, you should not use this function to resize array handles. Many platforms have memory alignment requirements that make it difficult to determine the correct size for the resulting array. Instead, you should use either `NumericArrayResize` or `SetCINArraySize`, which are described in the *Resizing Arrays and Strings* section of Chapter 2, *CIN Parameter Passing*. You should not use these functions on a locked handle.

Parameter	Type	Description
h	`UHandle`	Handle to resize.
size	`int32`	New size, in bytes, of the handle.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mFullErr`	Not enough memory to perform operation.
`mZoneErr`	Handle or pointer not in specified zone.

AZSetHSzClr
DSSetHSzClr

syntax

```
MgErr       ZSetHSzClr(h, size);
MgErr       DSSetHSzClr(h, size);
```

`XXSetHSzClr` changes the size of the block of memory referenced by the specified handle and sets any new memory to zero. You should not use this function on a locked handle.

Parameter	Type	Description
h	`UHandle`	Handle to resize.
size	`int32`	New size, in bytes, of the handle.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mFullErr`	Not enough memory to perform operation.
`mZoneErr`	Handle or pointer not in specified zone.

Allocating and Releasing Pointers

AZDisposePtr
DSDisposePtr

syntax

```
MgErr       AZDisposePtr(p);
MgErrD      SDisposePtr(p);
```

`XXDisposePtr` releases the memory referenced by the specified pointer.

Parameter	Type	Description
p	`UPtr`	Pointer to dispose.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mZoneErr`	Handle or pointer not in specified zone.

AZNewPClr
DSNewPClr

syntax

```
UPtr        AZNewPClr(size);
UPtr        DSNewPClr(size);
```

XXNewPClr creates a new pointer to a nonrelocatable block of memory of the specified size and initializes the memory to zero.

Parameter	Type	Description
size	int32	Size, in bytes, of the pointer to create.

returns A pointer to a block of **size** bytes filled with zeros. Returns NULL if the allocation could not be performed.

AZNewPtr
DSNewPtr

syntax

```
UPtr        AZNewPtr(size);
UPtr        DSNewPtr(size);
```

XXNewPtr creates a new pointer to a nonrelocatable block of memory of the specified size.

Parameter	Type	Description
size	int32	Size, in bytes, of the pointer to create.

returns A pointer to a block of **size** bytes. Returns NULL if the allocation could not be performed.

Manipulating Properties of Handles

AZHLock

syntax

```
MgErr       AZHLock(h);
```

AZHLock locks the memory referenced by the application zone handle **h** so that the memory cannot move. This means the memory manager cannot move the block of memory to which the handle refers.

Do not lock handles more than necessary; it interferes with efficient memory management. Also, do not enlarge a locked handle.

Parameter	Type	Description
h	UHandle	Application zone handle to lock.

returns MgErr, which can contain the errors in the following list.

Error	Description
noErr	No error.
mZoneErr	Handle or pointer not in specified zone.

AZHPurge

syntax `void AZHPurge(h);`

AZHPurge marks the memory referenced by the application zone handle **h** as purgeable. This means that in tight memory conditions the memory manager can perform an AZEmptyHandle on **h.** Use AZReallocHandle() to reuse a handle if the manager purges it.

If you mark a handle as purgeable, check the handle before using it to see if it has become an empty handle.

Parameter	Type	Description
h	UHandle	Application zone handle to mark as purgeable.

AZHNoPurge

syntax `void AZHNoPurge(h);`

AZHNoPurge marks the memory referenced by the application zone handle **h** as unpurgeable.

Parameter	Type	Description
h	UHandle	Application zone handle to mark as unpurgeable.

AZHUnlock

syntax `MgErr AZHUnlock(h);`

AZHUnlock unlocks the memory referenced by the application zone handle **h** so that it can be moved. This means that the memory manager can move the block of memory to which the handle refers if other memory operations need space.

Parameter	Type	Description
h	UHandle	Application zone handle to unlock.

returns MgErr, which can contain the errors in the following list.

Error	Description
noErr	No error.
mZoneErr	Handle or pointer not in specified zone.

Memory Utilities

AZHandAndHand
DSHandAndHand

syntax
```
MgErr       AZHandAndHand(h1, h2);
MgErr       DSHandAndHand(h1, h2);
```

XXHandAndHand appends the data referenced by **h1** to the end of the memory block referenced by **h2**.

The function resizes handle **h2** to hold **h1** and **h2** data. If **h1** is an AZ handle, you should lock it, because this routine can move memory.

Parameter	Type	Description
h1	UHandle	Source of data to append to **h2**.
h2	UHandle	Initial handle, to which the data of **h1** is appended.

returns MgErr, which can contain the errors in the following list.

Error	Description
noErr	No error.
mFullErr	Not enough memory to perform operation.
mZoneErr	Handle or pointer not in specified zone.

AZHandToHand
DSHandToHand

syntax

```
MgErr       AZHandToHand(hp);
MgErr       DSHandToHand(hp);
```

`XXHandToHand` copies the data referenced by the handle to which **hp** points into a new handle, and returns a pointer to the new handle in **hp**.

You can use this routine to copy an existing handle into a new handle. The old handle remains allocated. This routine writes over the pointer that is passed in, so you should maintain a copy of the original handle.

Parameter	Type	Description
hp	`UHandle *`	Pointer to handle to duplicate. A pointer to the resulting handle is returned in this parameter. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns

`MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mFullErr`	Not enough memory to perform operation.
`mZoneErr`	Handle or pointer not in specified zone.

AZPtrAndHand
DSPtrAndHand

syntax

```
MgErr       AZPtrAndHand(p, h, size);
MgErr       DSPtrAndHand(p, h, size);
```

`XXPtrAndHand` appends **size** bytes from the address referenced by **p** to the end of the memory block referenced by **h**.

Parameter	Type	Description
p	`UPtr`	Source of data to append to **h**.
h	`UHandle`	Handle to which the data of **p** is appended.
size	`int32`	Number of bytes to copy from **p**.

returns

`MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.

mFullErr	Not enough memory to perform operation.
mZoneErr	Handle or pointer not in specified zone.

AZPtrToHand
DSPtrToHand

syntax

```
MgErr       AZPtrToHand(p, hp, size);
MgErrD      SPtrToHand(p, hp, size);
```

XXPtrToHand creates a new handle of **size** bytes and copies **size** bytes from the address referenced by **p** to the handle.

Parameter	Type	Description
p	UPtr	Source of data to copy to the handle pointed to by **hp**.
hp	UHandle *	Pointer to new handle. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.
size	int32	Number of bytes to copy from **p** to the new handle.

returns MgErr, which can contain the errors in the following list.

Error	Description
noErr	No error.
mFullErr	Not enough memory to perform operation.

AZPtrToXHand
DSPtrToXHand

syntax

```
MgErr       AZPtrToXHand(p, h, size);
MgErr       DSPtrToXHand(p, h, size);
```

XXPtrToXHand copies **size** bytes from the address referenced by **p** to the existing handle **h**, resizing **h**, if necessary, to hold the results.

Parameter	Type	Description
p	UPtr	Source of data to copy to the handle **h**.
h	UHandle	Destination handle.
size	int32	Number of bytes to copy from **p** to the existing handle.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`noErr`	No error.
`mFullErr`	Not enough memory to perform operation.
`mZoneErr`	Handle or pointer not in specified zone.

ClearMem

syntax `void ClearMem(p, size);`

`ClearMem` sets **size** bytes starting at the address referenced by **p** to 0.

Parameter	Type	Description
p	`UPtr`	Pointer to block of memory to clear.
size	`int32`	Number of bytes to clear.

MoveBlock

syntax `void MoveBlock(ps, pd, size);`

`MoveBlock` moves **size** bytes from one address to another. The source and destination memory blocks can overlap.

Parameter	Type	Description
ps	`UPtr`	Pointer to source.
pd	`UPtr`	Pointer to destination.
size	`int32`	Number of bytes to move.

SwapBlock

syntax `void SwapBlock(ps, pd, size);`

`SwapBlock` swaps **size** bytes between the section of memory referred to by **ps** and **pd**. The source and destination memory blocks should not overlap.

Parameter	Type	Description
ps	`UPtr`	Pointer to source.
pd	`UPtr`	Pointer to destination.

size	int32	Number of bytes to move.

Handle and Pointer Verification

AZCheckHandle
DSCheckHandle

syntax

```
MgErr       AZCheckHandle(h);
MgErr       DSCheckHandle(h);
```

XXCheckHandle verifies that the specified handle is really a handle. If the handle is not a real handle, this function returns mZoneErr.

Parameter	Type	Description
h	UHandle	Handle to verify.

returns MgErr, which can contain the errors in the following list.

Error	Description
noErr	No error.
mZoneErr	Handle or pointer not in specified zone.

AZCheckPtr
DSCheckPtr

syntax

```
MgErr       AZCheckPtr(p);
MgErr       DSCheckPtr(p);
```

XXCheckPtr verifies that the specified pointer is a pointer allocated with XXNewPtr or XXNewPClr. If the pointer is not a real pointer, this function returns mZoneErr.

Parameter	Type	Description
p	UPtr	Pointer to verify.

returns MgErr, which can contain the errors in the following list.

Error	Description
noErr	No error.
mZoneErr	Handle or pointer not in specified zone.

Memory Zone Utilities

AZHeapCheck
DSHeapCheck

syntax

```
int32       AZHeapCheck(Bool32 d);
int32       DSHeapCheck(Bool32 d);
```

`XXHeapCheck` verifies that the specified heap is not corrupt. This function returns a zero for an intact heap and a nonzero value for a corrupt heap.

Parameter	Type	Description
d	`Bool32`	Dump extensive heap examination to auxiliary screen.

returns `int32`, which can contain the errors in the following list.

Value	Description
`noErr`	The heap is intact.
`mCorruptErr`	The heap is corrupt.

AZMaxMem
DSMaxMem

syntax

```
int32       AZMaxMem();
int32       DSMaxMem();
```

`XXMaxMem` returns the size of the largest block of contiguous memory available for allocation.

returns `int32`, the size of the largest block of contiguous memory available for allocation.

AZMemStats
DSMemStats

syntax

```
void        AZMemStats(MemStatRec *msrp);
void        DSMemStats(MemStatRec *msrp);
```

`XXMemStats` returns various statistics about the memory in a zone.

Parameter	Type	Description
msrp	`MemStatRec*`	Returns statistics about the zone's free memory in a `MemStatRec` structure. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

A `MemStatRec` structure is defined as follows.

```
typedef struct {
   int32 totFreeSize, maxFreeSize, nFreeBlocks;
   int32 totAllocSize, maxAllocSize;
   int32 nPointers, nUnlockedHdls, nLockedHdls;
   int32 reserved [4];
   }
```

The free memory in a zone consists of a number of blocks of contiguous memory. In the `MemStatRec` structure, **totFreeSize** is the sum of the sizes of these blocks, **maxFreeSize** is the largest of these blocks (as returned by `XXMaxMem`), and **nFreeBlocks** is the number of these blocks.

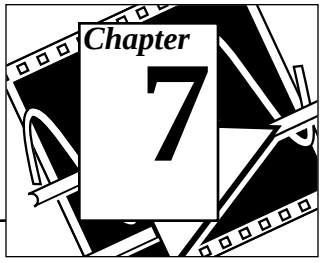
Similarly, the allocated memory in a zone consists of a number of blocks of contiguous memory. In the `MemStatRec` structure, **totAllocSize** is the sum of the sizes of these blocks and **maxAllocSize** is the largest of these blocks.

Because there are three different varieties of allocated blocks, the numbers of blocks of each type is returned separately.

nPointers (`int32`) is the number of pointers. **nUnlockedHdls** (`int32`) is the number of unlocked handles. **nLockedHdls** (`int32`) is the number of locked handles. Add these three values together to find the total number of allocated blocks.

The four **reserved** fields are reserved for use by National Instruments.

File Manager Functions



File Manager Data Structures

File/Directory Information Record

Several routines in the file manager work with a data structure that defines the attributes of a file or directory. The following list gives the file/directory information record.

```
typedef struct {
    int32     type;         * system specific file type-
                            - 0 for directories */
    int32     creator;      * system specific file
                            creator-- 0 for folders (on
                            Mac only)*/
    int32     permissions;  * system specific file access
                            rights */
    int32     size;         /* file size in bytes (data
                            fork on Mac) or entries in
                            directory*/
    int32     rfSize;       /* resource fork size (on Mac
                            only) */
    uInt32    cdate;        /* creation date: seconds
                            since system reference time
                            */
    uInt32    mdate;        /* last modification date:
                            seconds since system ref time
                            */
    Bool32    folder;       /* indicates whether path
                            refers to a folder */
    Bool32    isInvisible;  /* indicates whether file is
                            visible in File Dialog (on
                            Mac only)*/
```

```
    Point     location;     /* system specific desktop
                            geographical location (on Mac
                            only)*/
    Str255    owner;        /* owner (in pascal string
                            form) of file or folder */
    Str255    group;        /* group (in pascal string
                            form) of file or folder */
    }         FInfoRec, *FInfoPtr;
```

File Type Record

The file type record is:

```
typedef struct {
  int32   flags;
  int32   type;
}         FileType;
```

Only the least significant four bits of `flags` contain useful information. The remaining bits are reserved for use by LabVIEW. You can test these four bits using the following four masks:

```
#define kIsFile 0x01
#define kRecognizedType 0x02
#define kIsLink 0x04
#define kFIsInvisible 0x08
```

The `kIsFile` bit is set if the item described by the file type record is a file; otherwise it is clear. The `kRecognizedType` bit is set if the item described is a file for which you can determine a 4-character file type; otherwise it is clear. The `kIsLink` bit is set if the item described is a UNIX link or Macintosh alias; otherwise it is clear. The `kFIsInvisible` bit is set if the item described will not appear in a file dialog; otherwise it is clear.

The value of `type` is defined only if the `kRecognizedType` bit is set in `flags`. In this case, `type` is the 4-character file type of the file described by the file type record. This 4-character file type is provided by the file system on the Macintosh and is computed by examining the file name extension on other systems.

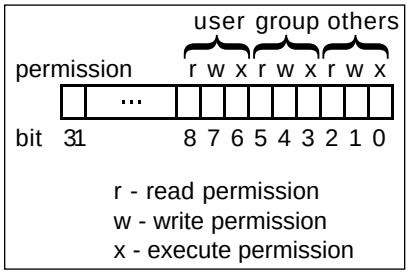
Path Data Type

The file manager defines the `Path` data type for use in describing paths to files and directories. The data structure for the `Path` data type is private. You use file manager routines to create and manipulate `Path`s.

Permissions

The file manager uses the `int32` data type to describe permissions for files and directories. The manager uses only the least significant nine bits of the `int32`.

On a UNIX computer, the nine bits of permissions correspond exactly to nine UNIX permission bits governing read, write, and execute permissions for user, group, and others. Permission bits on a UNIX system are represented in the following illustration.



On the PC, permissions are ignored for directories. For files, only bit 7 (the UNIX user write permission bit) is used. If this bit is clear, the file is read-only. Otherwise, you can write to the file.

On the Macintosh, all nine bits are used for directories (folders). The bits which control read, write, and execute permissions, respectively, on a UNIX system are used to control See Files, Make Changes, and See Folders access rights, respectively, on the Macintosh. For files, only bit 7 (the UNIX user write permission bit) is used. If this bit is clear, the file is locked. Otherwise, the file is not locked.

Volume Information Record

The volume information record is:

```
typedef struct {
    int32   size;    /* size in bytes of a kuhvkjhgvku
                        volume */
    int32   used;    /* number of bytes used on volume
                        */
    int32   free;    /* number of bytes available for
                        use on volume */
}           VInfoRec;
```

File Manager Functions

Performing Basic File Operations

FCreate

syntax `MgErr FCreate(fdp, path, permissions, openMode, denyMode, group);`

`FCreate` creates a file with the name and location specified by **path** and with the specified **permissions**, and opens it for writing and reading, as specified by **openMode**. If the file already exists, an error is returned.

You can use **denyMode** to control concurrent access to the file from within LabVIEW. The **group** parameter allows you to assign the file to a UNIX group; under Windows or Macintosh, **group** is ignored.

If the function creates the file, the resulting file descriptor is stored in the address referred to by **fdp**. If an error occurs, the function stores `0` in the address referred to by **fdp** and returns an error.

Note: ***Before attempting to call this function, make sure that you understand how to use the* fdp *parameter. See the* Pointers as Parameters *section of Chapter 1,* CIN Overview*, for more information about this parameter.***

Parameter	Type	Description
fdp	`File *`	Address at which `FCreate` stores the file descriptor for the new file. If `FCreate` fails, it stores `0` in the address **fdp**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.
path	`Path`	Path of the file that you want to create.

permissions	`int32`	Permissions to assign to the new file. See the *File Manager Data Structures* section for a description of **permissions**.
openMode	`int32`	Access mode to use in opening the file. Can have the following values, which are defined in the file `extcode.h`. • `openReadOnly`: Open for reading. • `openWriteOnly`: Open for writing • `openReadWrite`: Open for both reading and writing
denyMode	`int32`	Mode that determines what level of concurrent access to the file is allowed. Can have the following values, which are defined in the file `extcode.h`. • `denyReadWrite`: Prevents others from reading from and writing to the file while it is open. • `denyWriteOnly`: Prevents others from writing to the file only while it is open • `denyNeither`: allows others to read from and write to the file while it is open.
group	`PStr`	UNIX group you want to assign to the new file.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fIsOpen`	File is already open for writing. This error is returned only on the Macintosh and the Sun. The PC returns `fIOErr` when the file is already open for writing.
`fNoPerm`	Access denied (something is locked/protected).
`fDupPath`	A file of that name already exists.
`fTMFOpen`	Too many files open.
`fIOErr`	Unspecified I/O error occurred.

FCreateAlways

syntax

```
MgErr       FCreateAlways(fdp, path, permissions,
                          openMode, denyMode, group);
```

`FCreateAlways` creates a file with the name and location specified by **path** and with the specified **permissions**, and opens the file for writing and reading, as specified by **openMode**. If the file already exists, this function opens and truncates the file.

You can use **denyMode** to control concurrent access to the file from within LabVIEW. The **group** parameter allows you to assign the file to a UNIX group; under Windows or Macintosh, **group** is ignored.

If the function creates the file, the resulting file descriptor is stored in the address referred to by **fdp**. If an error occurs, the function stores `0` in the address referred to by **fdp** and returns an error.

Note: ***Before attempting to call this function, make sure that you understand how to use the* fdp *parameter. See the* Pointers as Parameters *section of Chapter 1,* CIN Overview*, for more information about this parameter.***

Parameter	Type	Description
fdp	`File *`	Address at which `FCreateAlways` stores the file descriptor for the new file. If `FCreateAlways` fails, it stores 0 in the address **fdp**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.
path	`Path`	Path of the file that you want to create.
permissions	`int32`	Permissions to assign to the new file. See the *File Manager Data Structures* section of this chapter for a description of **permissions**.
openMode	`int32`	See FMOpen for a description of **openMode**.
denyMode	`int32`	See FMOpen for a description of **denyMode**.
group	`PStr`	UNIX group you want to assign to the new file.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fIsOpen`	File is already open for writing. This error is returned only on the Macintosh and the Sun. The PC returns `fIOErr` when the file is already open for writing.
`fNoPerm`	Access denied (something is locked/protected).
`fDupPath`	A file of that name exists.
`fTMFOpen`	Too many files open.
`fIOErr`	Unspecified I/O error occurred.

FMClose

syntax `MgErr FMClose(fd);`

`FMClose` closes the file associated with the file descriptor **fd**.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file you want to close.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	Not a valid file descriptor.
`fIOErr`	Unspecified I/O error occurred.

FMOpen

syntax `MgErr FMOpen(fdp, path, openMode, denyMode);`

Note: ***Before attempting to call this function, make sure that you understand how to use the* fdp *parameter. See the* Pointers as Parameters *section of Chapter 1,* CIN Overview*, for more information about this parameter.***

`FMOpen` opens a file with the name and location specified by **path** for writing and reading, as specified by **openMode**.

With the **denyMode** parameter, you control concurrent access to the file from within LabVIEW.

If this function opens the file, the resulting file descriptor is stored in the address referred to by **fdp**. If an error occurs, `0` is stored in the address referred to by **fdp** and the error is returned.

Parameter	Type	Description
fdp	`File *`	Address at which `FMOpen` stores the file descriptor for the opened file. If the function fails, `FMOpen` stores 0 in the address **fdp**.

		See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.
path	`Path`	Path of the file that you want to open.
openMode	`int32`	Access mode to use in opening the file. Can have the following values, which are defined in the file `extcode.h`. • `openReadOnly`: Open for reading. • `openWriteOnly`: Open for writing; file is not truncated (data is not removed). On the Macintosh, this mode provides true write-only access to files. On a PC or a UNIX system, LabVIEW I/O functions are built in the C standard I/O library, with which you have write-only access to a file only if you are truncating the file or making the access append-only. Therefore, this mode actually allows both read and write access to files on a PC or UNIX system. • `openReadWrite`: Open for both reading and writing. • `openWriteOnlyTruncate`: Open for writing; truncates the file.
denyMode	`int32`	Mode that determines what level of concurrent access to the file is allowed. Can have the following values, which are defined in the file `extcode.h`. • `denyReadWrite`: Prevents others from reading from and writing to the file while it is open. • `denyWriteOnly`: Prevents others from writing to the file only while it is open • `denyNeither`: allows others to read from and write to the file while it is open.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.

fIsOpen	File is already open for writing. This error is returned only on the Macintosh and the Sun. The PC returns fIOErr when the file is already open for writing.
fNotFound	File not found.
fTMFOpen	Too many files open.
fIOErr	Unspecified I/O error occurred.

FMRead

syntax `MgErr FMRead(fd, inCount, outCountp, buffer);`

`FMRead` reads **inCount** bytes from the file specified by the file descriptor **fd**. The function starts from the current position mark (see the `FSeek` and `FTell` functions), and reads the data into memory, starting at the address specified by **buffer**.

The function stores the actual number of bytes read in ***outCountp**. The number of bytes can be less than **inCount** if the function encounters end-of-file before reading **inCount** bytes. The number of bytes will be zero if any other error occurs.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file from which you want to read.
inCount	`int32`	Number of bytes you want to read.
outCountp	`int32 *`	Address at which `FMRead` stores the number of bytes read. `FMRead` will not store any value if `NULL` is passed. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.
buffer	`UPtr`	Address where `FMRead` will store the data.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
mgArgErr	Not a valid file descriptor or **inCount** < 0.
fEOF	EOF encountered.
fIOErr	Unspecified I/O error occurred.

FMWrite

syntax `MgErr FMWrite(fd, inCount, outCountp, buffer);`

`FMWrite` writes **inCount** bytes from memory, starting at the address specified by **buffer**, to the file specified by the file descriptor **fd**, starting from the current position mark (see the `FSeek` and `FTell` functions).

The function stores the actual number of bytes written in ***outCountp**. The number of bytes stored can be less than **inCount** if an `fDiskFull` error occurs before the function writes **inCount** bytes. The number of bytes stored will be zero if any other error occurs.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file to which you want to write.
inCount	`int32`	Number of bytes you want to write.
outCountp	`int32 *`	Address at which `FMWrite` stores the number of bytes actually written. `FMWrite` will not store any value if `NULL` is passed. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.
buffer	`UPtr`	Address of the data you want to write.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	Not a valid file descriptor or **inCount** < 0.
`fDiskFull`	Out of space.
`fNoPerm`	Access denied.
`fIOErr`	Unspecified write error occurred.

Positioning the Current Position Mark

FMSeek

syntax `MgErr FMSeek(fd, ofst, mode);`

`FMSeek` sets the current position mark for a file to the specified point, relative to the beginning of the file, the current position in the file, or the end of the file. If an error occurs, the current position mark does not move.

Parameter	Type	Description

fd	File	File descriptor associated with the file.
ofst	int32	New position of the current position mark. The position is the number of bytes from the beginning of the file, the current position mark, or the end of the file, as determined by **mode**.
mode	int32	Position in the file relative to which FMSeek sets the current position mark for a file. If **mode** is fStart, the current position mark moves to **ofst** bytes relative to the start of the file (**ofst** must be greater than or equal to 0). If **mode** is fCurrent, the current position mark moves **ofst** bytes from the current position mark (**ofst** can be positive, 0, or negative). If **mode** is fEnd, the current position mark moves to **ofst** bytes from the end of the file (**ofst** must be less than or equal to 0).

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	Not a valid file descriptor.
fEOF	Attempt to seek before the start or after the end of the file.
fIOErr	Unspecified I/O error occurred.

FMTell

syntax `MgErr FMTell(fd, ofstp);`

FMTell returns the position of the current position mark in the file.

Parameter	Type	Description
fd	File	File descriptor associated with the file.
ofstp	int32 *	Address at which FMTell stores the position of the current position mark, in terms of bytes relative to the beginning of the file. If an error occurs, the contents of **ofstp** is undefined. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	Not a valid file descriptor.
fIOErr	Unspecified I/O error occurred.

Positioning the End-Of-File Mark

FGetEOF

syntax `MgErr FGetEOF(fd, sizep);`

`FGetEOF` returns the size of the specified file.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file.
sizep	`int32 *`	Address at which `FGetEOF` stores the size of the file in bytes. If an error occurs, the contents of ***sizep** is undefined. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
mgArgErr	Not a valid file descriptor.
fIOErr	Unspecified I/O error occurred.

FSetEOF

syntax `MgErr FSetEOF(fd, size);`

`FSetEOF` sets the size of the specified file. If an error occurs, the file size does not change.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file.
size	`int32`	New file size in bytes.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
mgArgErr	Not a valid file descriptor or **size** < 0.
fDiskFull	Disk is full.
fNoPerm	Access denied (file exists or something is locked/protected).
fIOErr	Unspecified I/O error occurred.

Flushing File Data to Disk

FFlush

syntax `MgErr FFlush(fd);`

`FFlush` writes any buffered data for the specified file out to the disk.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
mgArgErr	Not a valid file descriptor.
fIOErr	Unspecified I/O error occurred.

Determining File, Directory, and Volume Information

FExists

syntax `int32 FExists(path);`

`FExists` returns information about the specified file or directory. It returns less information than FGetInfo, but it is much quicker on many platforms.

Parameter	Type	Description
path	`Path`	Path of the file or directory about which you want information.

returns `int32`, which is one of the following values.

Error	Description
kFIsFile	Specified item is a file.

kFIsFolder	Specified item is a directory or folder.
kFNotExist	Specified item does not exist.

FGetAccessRights

syntax `MgErr FGetAccessRights(path, owner, group, permPtr);`

`FGetAccessRights` returns access rights information about the specified file or directory.

Parameter	Type	Description
path	`Path`	Path of the file or directory about which you want access rights information.
owner	`PStr`	Address at which `FGetAccessRights` stores the owner of the file or directory.
group	`PStr`	Address at which `FGetAccessRights` stores the group of the file or directory.
permPtr	`int32 *`	Address at which `FGetAccessRights` stores the permissions of the file or directory. See the *File Manager Data Structures* section of this chapter for a description of permissions. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fNotFound`	File not found.
`fIOErr`	Unspecified I/O error occurred.

FGetInfo

syntax `MgErr FGetInfo(path, infop);`

`FGetInfo` returns information about the specified file or directory.

Parameter	Type	Description

path	`Path`	Path of the file or directory about which you want information.
infop	`FInfoPtr`	Address where `FGetInfo` stores information about the file or directory. If an error occurs, the information is undefined. See the *File Manager Data Structures* section of this chapter for a description of the `FInfoPtr` data type. See also the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fNotFound`	File not found.
`fIOErr`	Unspecified I/O error occurred.

FGetVolInfo

syntax `MgErr FGetVolInfo(path, vinfo);`

`FGetVolInfo` gets a path specification and information for the volume containing the specified file or directory.

Parameter	Type	Description
path	`Path`	Path of a file or directory contained on the volume from which you want to get information. This path is overwritten with a path specifying the volume containing the specified file or directory. If an error occurs, this path is undefined.
vinfo	`VInfoRec *`	Address at which `FGetVolInfo` stores the information about the volume. If an error occurs, the information is undefined. See the *File Manager Data Structures* section of this chapter for a description of the `VInfoRec` data type. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fIOErr	Unspecified I/O error occurred.

FSetAccessRights

syntax `MgErr FSetAccessRights(path, owner, group, permPtr);`

`FSetAccessRights` sets access rights information for the specified file or directory. If an error occurs, no information changes.

Parameter	Type	Description
path	`Path`	Path of the file or directory for which you want to set access rights information.
owner	`PStr`	New owner that `FSetAccessRights` sets for the file or directory if **owner** is not `NULL`.
group	`PStr`	New group that `FSetAccessRights` sets for the file or directory if **group** is not `NULL`.
permPtr	`int32 *`	Address of new permissions that `FSetAccessRights` sets for the file or directory if **permPtr** is not `NULL`.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNotFound	File not found.
fIOErr	Unspecified I/O error occurred.

FSetInfo

syntax `MgErr FSetInfo(path, infop);`

`FSetInfo` sets information for the specified file or directory. If an error occurs, no information changes.

Parameter	Type	Description
path	`Path`	Path of the file or directory for which you want to set information.

infop	FInfoPtr	Address of information FSetInfo sets for the file or directory. See the *File Manager Data Structures* section of this chapter for a description of the FInfoPtr data type.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNotFound	File not found.
fIOErr	Unspecified I/O error occurred.

Getting Default Access Rights Information

FGetDefGroup

syntax `LStrHandle FGetDefGroup(groupHandle);`

FGetDefGroup gets the LabVIEW default group for a file or directory.

Parameter	Type	Description
groupHandle	LStrHandle	Handle that represents the LabVIEW default group for a file or directory. If **groupHandle** is NULL, FGetDefGroup allocates a new handle and returns the default group in it. If **groupHandle** is a handle, FGetDefGroup returns it, and **groupHandle** resizes to hold the default group.

returns The resulting LStrHandle; if **groupHandle** was not NULL, then the return value is the same LStrHandle as **groupHandle**. If an error occurs, NULL is returned.

Creating and Determining the Contents of Directories

FListDir

syntax `MgErr FListDir(path, list, typeH);`

FListDir determines the contents of a directory.

The function fills the (AZ) handle passed in **list** with a CPStr, where the **cnt** field specifies the number of concatenated Pascal strings that follow in the str[] field. See the *Dynamic Data Types* section of Chapter 5, *Manager Overview*, for a description of the CPStr data type. If **typeH** is not NULL, the function fills the AZ handle passed in **typeH** with the file type information for each file name or directory name stored in **list**.

Parameter	Type	Description
path	Path	Path of the directory whose contents you want to determine.
list	CPStrHandle	Application zone handle in which FListDir stores a series of concatenated Pascal strings, preceded with a 4-byte integer field, **cnt**, that indicates the number of items in the buffer.
typeH	FileType	Application zone handle in which FListDir stores a series of FileType records. If **typeH** is not NULL, then FListDir stores one FileType record in **typeH** for each Pascal string in list. The *n*th FileType in **typeH** denotes the file type information about the file or directory named in the *n*th string in **list**. See the *File Manager Data Structures* section of this chapter for a description of the FileType data type.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNotFound	Directory not found.
fNoPerm	Access denied (file/directory/disk is locked/protected).
mFullErr	Insufficient memory.
fIOErr	Unspecified I/O error occurred.

FNewDir

syntax MgErr FNewDir(path, permissions);

FNewDir creates a new directory with the specified **permissions**. If an error occurs, the function does not create the directory.

Parameter	Type	Description
path	Path	Path of the directory you want to create.

permissions	int32	Permissions for the new directory. See the *File Manager Data Structures* section of this chapter for a description of **permissions**.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNoPerm	Access denied (file/directory/disk is locked /protected).
fDupPath	Directory already exists.
fIOErr	Unspecified I/O error occurred.

Copying Files

FCopy

syntax MgErr FCopy(oldPath, newPath);

FCopy copies a file, preserving the type, creator, and access rights. The file to be copied must not be open. If an error occurs, the new file is not created.

Parameter	Type	Description
oldPath	Path	Path of the file you want to copy.
newPath	Path	Path, including filename, where you want the new file to be stored.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNotFound	The original file could not be found.
fNoPerm	Access denied (file/directory/disk is locked/protected).
fDiskFull	Disk is full.
fDupPath	The new file already exists.
fIsOpen	The original file is open for writing.
fTMFOpen	Too many files open.
mFullErr	Insufficient memory.
fIOErr	Read, write, or unspecified I/O error occurred

Moving and Deleting Files and Directories

FMove

syntax `MgErr FMove(oldPath, newPath);`

`FMove` moves a file or renames it if the new path indicates the file is to remain in the same directory.

Parameter	Type	Description
oldPath	`Path`	Path of the file or directory you want to move.
newPath	`Path`	Path, including the name of the file or directory, where you want the file or directory to be moved.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fNotFound`	The original file could not be found.
`fNoPerm`	Access denied (file/directory/disk is locked/protected).
`fDiskFull`	Disk is full.
`fDupPath`	The new file already exists.
`fIsOpen`	The original file is open for writing.
`fTMFOpen`	Too many files open.
`mFullErr`	Insufficient memory.
`fIOErr`	Read, write, or unspecified I/O error occurred.

FRemove

syntax `MgErr FRemove(path);`

`FRemove` deletes a file or a directory. If an error occurs, this function does not remove the file or directory.

Parameter	Type	Description
path	`Path`	Path of the file or directory you want to delete.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.

fNotFound	The file could not be found.
fNoPerm	Access denied (file/directory/disk is locked/protected).
fIsOpen	File is open or directory is not empty.
fIOErr	Unspecified I/O error occurred.

Locking a File Range

FLockOrUnlockRange

syntax MgErr FLockOrUnlockRange(fd, mode, offset, count, lock);

FLockOrUnlockRange locks or unlocks a section of a file.

Parameter	Type	Description
fd	File	File descriptor associated with the file.
mode	int32	Position in the file relative to which FLockOrUnlockRange determines the first byte to lock or unlock. If **mode** is fStart, the first byte to lock or unlock is located **offset** bytes from the start of the file (**offset** must be greater than or equal to 0). If **mode** is fCurrent, the first byte to lock or unlock is located **offset** bytes from the current position mark (**offset** can be positive, 0, or negative). If **mode** is fEnd, the first byte to lock or unlock is located **offset** bytes from the end of the file (**offset** must be less that or equal to 0).
offset	int32	The position of the first byte to lock or unlock. The position is the number of bytes from the beginning of the file, the current position mark, or the end of the file, as determined by **mode**.
count	int32	Number of bytes to lock or unlock starting at the location specified by **mode** and **offset**.
lock	Bool32	A boolean that specifies whether FLockOrUnlockRange locks or unlocks a range of bytes. If lock is TRUE this functions locks a range; if FALSE the function unlocks a range.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`fIOErr`	Unspecified I/O error occurred.

Matching Filenames with Patterns

FStrFitsPat

syntax `Bool32 FStrFitsPat(pat, str, pLen, sLen);`

`FStrFitsPat` determines whether a filename, **str**, matches a pattern, **pat**.

Parameter	Type	Description
pat	`uChar *`	Pattern (string) to which filename is to be compared. The following characters have special meanings in the pattern. • `\` : The following character is literal, not treated as having a special meaning. A single backslash at the end of **pat** is the same as two backslashes. • `?` : Match any one character. • `*` : Match zero or more characters.
str	`uChar *`	Filename (string) to compare to pattern.
pLen	`int32`	Number of characters in **pat**.
sLen	`int32`	Number of characters in **str**.

returns `FStrFitsPat` returns `TRUE` if the filename fits the pattern; `FALSE` if otherwise.

Creating Paths

FAddPath

syntax `MgErr FAddPath(basePath, relPath, newPath);`

`FAddPath` creates an absolute path by appending a relative path to an absolute path

☞ **Note:** ***You can pass in the same path variable for the new path that you use for the*** `basePath` ***or*** `relPath`***. Thus, the following three variations for calling this function work.***

```
FAddPath(basePath, relPath, newPath);
/* the new path is returned in a third path variable */
FAddPath(path, relPath, path);
/* the new path writes over the old base path */
FAddPath(basepath, path, path);
/* the new path writes over the old relative path */
```

Parameter	Type	Description
basePath	`Path`	Absolute path to which you want to append a relative path.
relPath	`Path`	Relative path you want to append to the existing base path.
newPath	`Path`	Path returned by `FAddPath`.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.

FAppendName

syntax `MgErr FAppendName(path, name);`

`FAppendName` appends a file or directory name to an existing path.

Parameter	Type	Description
path	`Path`	Base path to which you want to append a new file or directory name. `FAppendName` returns the resulting **path** in this parameter.
name	`PStr`	File or directory name that you want to append to the existing path.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
mFullErr	Insufficient memory.

FAppPath

syntax `MgErr FAppPath(p);`

FAppPath determines the path to the currently executing LabVIEW application.

Parameter	Type	Description
p	Path	Path in which FAppPath stores the path to the currently executing LabVIEW application. **p** must already be an allocated path.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
mFullErr	Insufficient memory.
fNotFound	File not found.
fIOErr	Unspecified I/O error occurred.

FEmptyPath

syntax `Path FEmptyPath(p);`

FEmptyPath makes an empty absolute path. Making a path an empty absolute path is not the same as disposing the path.

Parameter	Type	Description
p	Path	Path allocated by FEmptyPath. If **p** is NULL, FEmptyPath allocates a new path and returns the value. If **p** is a path, the existing path is set to be an empty path, and the new **p** is returned.

returns The resulting path; if **p** was not NULL, the return value is the same empty absolute path as **p**. If an error occurs, NULL is returned.

FMakePath

syntax `Path FMakePath(path, type, [volume, directory, directory, ..., name,] NULL);`

The brackets indicate that the **volume**, **directory**, and **name** parameters are optional.

`FMakePath` creates a new path. If **path** is `NULL`, the function allocates and returns a new path. Otherwise, **path** is set to the new path, and **path** is returned. If an error occurs, or the path is not specified correctly, `NULL` is returned.

When you are finished using a path, you should dispose of it using `FDisposePath`.

Parameter	Type	Description
path	`Path`	Parameter in which `FMakePath` returns the newly created path if **path** is not `NULL`.
type	`int32`	Type of path to create. If **type** is `fAbsPath`, the new path will be absolute. If **type** is `fRelPath`, the new path will be relative.
vol	`PStr`	Pascal string containing a legal volume name. An empty string means go up a level in the path hierarchy. This parameter is optional, and is only used for absolute paths on Macintosh or Windows platforms.
directory	`PStr`	Pascal string containing a legal directory name. An empty string means go up a level in the path hierarchy. Parameter is optional.
name	`PStr`	File or directory name. An empty string means go up a level in the path hierarchy. Parameter is optional.
NULL	`PStr`	Marker indicating the end of the path.

returns The resulting **path**; if you specified **path**, the return value is the same path as **path**. If an error occurs, **NULL** is returned.

FNotAPath

syntax `Path FNotAPath(p);`

`FNotAPath` creates a path that is the canonical invalid path.

Parameter	Type	Description
p	`Path`	Path allocated by `FNotAPath`. If **p** is `NULL`, `FNotAPath` allocates a new canonical invalid path

and returns the value. If **p** is a path, the existing path is set to the canonical invalid path, and the new **p** is returned.

returns The resulting path. If **p** was not NULL, the return value is the same canonical invalid path as **p**. If an error occurs, NULL is returned.

FRelPath

syntax `MgErr FRelPath(startPath, endPath, relPath);`

`FRelPath` computes a relative path between two absolute paths.

Note: ***You can pass in the same path variable for the new path that you use for the*** `startPath` ***or*** `relPath`***. Thus, the following three variations for calling this function work.***

```
FRelPath(startPath, endPath, relPath);
/* the relative path is returned in a third path variable */
FRelPath(startPath, endPath, startPath);
/* the new path writes over the old startPath */
FRelPath(startPath, endPath, endPath);
/* the new path writes over the old endPath */
```

Parameter	Type	Description
startPath	Path	Absolute path from which you want the relative path to be computed.
endPath	Path	Absolute path to which you want the relative path to be computed.
relPath	Path	Path returned by fAddPath.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
mFullErr	Insufficient memory.

Disposing Paths

FDisposePath

syntax `MgErr FDisposePath(p);`

`FDisposePath` disposes of the specified path.

Parameter	Type	Description
p	`Path`	Path you want to dispose of.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mZoneErr`	Invalid path**.**

Duplicating Paths

FPathCpy

syntax `MgErr FPathCpy(dst, src);`

`FPathCpy` duplicates the path specified by **src**, and stores the resulting path in the existing path, **dst**.

Parameter	Type	Description
dst	`Path`	Path where `FPathCpy` places the resulting duplicate path. This path must already have been created.
src	`Path`	Path that you want to duplicate.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.

FPathToPath

syntax `MgErr FPathToPath(p);`

FPathToPath duplicates the specified path and returns the new path in the same variable.

Parameter	Type	Description
p	Path *	Address of path to duplicate. Variable to which FPathToPath returns the resulting path. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.

Extracting Information from a Path

FDepth

syntax int32 FDepth(path);

FDepth computes the depth (number of component names) of a specified path.

Parameter	Type	Description
path	Path	Path whose depth you want to determine.

returns int32 indicating the depth of the specified path, which can have the following values for this function.

Value	Description
-1	Badly formed path.
0	Path is the root directory.
1	Path is in the root directory.
2	Path is in a subdirectory of the root directory, one level from the root directory.
n-1	Path is *n-2* levels from the root directory.
n	Path is *n-1* levels from the root directory.

FDirName

syntax `MgErr FDirName(path, dir);`

`FDirName` creates a path for the parent directory of a specified path.

Note: ***You can pass in the same path variable for the parent path that you use for*** `path`***. Thus, the following variations for calling this function work.***

```
err = FDirName(path, dir);
/* the parent path is returned in a second path variable */
err = FDirName(path, path);
/* the parent path writes over the existing path */
```

Parameter	Type	Description
path	`Path`	Path whose parent path you want to determine.
dir	`Path`	Parameter in which `FDirName` stores the parent path.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.

FName

syntax `MgErr FName(path, name);`

`FName` copies the last component name of a specified path into a string handle and resizes the handle as necessary.

Parameter	Type	Description
path	`Path`	Path whose last component name you want to determine.
name	`StringHandle`	Handle in which `FName` returns the last component name as a Pascal string.

returns `MgErr`, which can contain the errors in the following list.

Error	Description

mgArgErr	Badly formed path or path is root directory.
mFullErr	Insufficient memory.

FNamePtr

syntax `MgErr FNamePtr(path, name);`

`FNamePtr` copies the last component name of a specified path to the address specified by **name**. This routine does not allocate space for the returned data, so **name** must specify allocated memory of sufficient size to hold the component name.

Parameter	Type	Description
path	`Path`	Path whose last component name you want to determine.
name	`PStr`	Address at which `FNamePtr` stores the last component name as a Pascal string. This address must specify allocated memory of sufficient size to hold the name. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
mgArgErr	Badly formed path or path is root directory.
mFullErr	Insufficient memory.

FVolName

syntax `MgErr FVolName(path, vol);`

`FVolName` creates a path for the volume of a specified absolute path by removing all but the first component name from **path**.

Note: ***You can pass in the same path variable for the volume path that you use for*** `path`***. Thus, the following variations for calling this function work.***

```
err = FVolName(path, vol);
/* the parent path is returned in a second path variable */
```

```
err = FVolName(path, path);
/* the parent path writes over the existing path */
```

Parameter	Type	Description
path	Path	Path whose volume path you want to determine.
vol	Path	Parameter in which FVolName stores the volume path.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.

Converting Paths to and from Other Representations

FArrToPath

syntax MgErr FArrToPath(arr, relative, path);

FArrToPath converts a specified one-dimensional LabVIEW array of strings to a path of the type specified by **relative**. Each string in the specified array is converted in order into a component name of the resulting path.

If no error occurs, **path** is set to a path whose component names are the strings in **arr**. If an error occurs, **path** is set to the canonical invalid path.

Parameter	Type	Description
arr	UHandle	The (DS) handle containing the array of strings which you wish to convert to a path.
relative	Bool32	If **relative** is TRUE, then the resulting path is relative; otherwise, the resulting path is absolute.
path	Path	Path where FArrToPath stores the resulting path. This path must already have been allocated.

returns MgErr, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
mFullErr	Insufficient memory.

FFlattenPath

syntax `int32 FFlattenPath(p, fp);`

`FFlattenPath` converts a path into a flat form that you can use to write the path as information to a file. The function stores the resulting flat path in a pre-allocated buffer and returns the number of bytes.

You can determine the size needed for the flattened path by passing `NULL` for **fp**, in which case the function returns the necessary size without writing anything into the location pointed to by **fp**.

Parameter	Type	Description
p	`Path`	Path you want to flatten.
fp	`UPtr`	Address in which `FFlattenPath` stores the resulting flattened path. If this value is `NULL`, `FFlattenPath` does not write anything to this address, but does return the size that the flattened path would require. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `int32`, indicating the number of bytes required to store the flattened path.

FPathToArr

syntax `MgErr FPathToArr(path, relativePtr, arr);`

`FPathToArr` converts a specified path to a one-dimensional LabVIEW array of strings and determines whether the specified path is relative. Each component name of the specified path is converted in order into a string in the resulting array.

If no error occurs, **arr** is set to an array of strings containing the component names of **path**. If an error occurs, **arr** is set to an empty array.

Parameter	Type	Description
path	`Path`	The path which you wish to convert to an array of strings.
relativePtr	`Bool32 *`	Address at which to store a boolean value telling whether the specified path is relative.

		See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.
arr	`UHandle`	(DS) Handle where `FPathToArr` stores the resulting array of strings. This handle must already have been allocated.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	Badly formed path or unallocated array.
`mFullErr`	Insufficient memory.

FPathToAZString

syntax `MgErr FPathToAZString(p, txt);`

`FPathToAZString` converts a specified path to an `LStr` and stores the string as an application zone handle. The `LStr` contains the platform-specific syntax for the path.

Parameter	Type	Description
p	`Path`	Path that you want to convert to a string.
txt	`LStrHandle *`	Address at which `FPathToAZString` stores the resulting string. If the value at **txt** is nonzero, the function assumes that it is a valid handle, resizes the handle, fills in its value, and stores the handle at the address referred to by **txt**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error occurred.

FPathToDSString

syntax `MgErr FPathToDSString(p, txt);`

`FPathToDSString` converts a specified path to an `LStr` and stores the string as a data space zone handle. The `LStr` contains the platform-specific syntax for the path.

Parameter	Type	Description
p	`Path`	Path that you want to convert to a string.
txt	`LStrHandle *`	Address at which `FPathToDSString` stores the resulting string. If the value at **txt** is nonzero, the function assumes that it is a valid handle, resizes the handle, fills in its value, and stores the handle at the address referred to by **txt**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error occurred.

FStringToPath

syntax `MgErr FStringToPath(text, p);`

`FStringToPath` creates a path from an `LStr`. The `LStr` contains the platform-specific syntax for a path.

Parameter	Type	Description
text	`LStrHandle`	String that contains the path in platform-specific syntax.
p	`Path *`	Address at which `FStringToPath` stores the resulting path. If the value at **p** is non-zero, the function assumes that it is a valid path, resizes the path, and fills in its value. If the value at **p** is zero (NULL), the function creates a new path, fills in its value, and stores the path at the address referred to by **p**.

See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mFullErr`	Insufficient memory.

FTextToPath

syntax `MgErr FTextToPath(text, tlen, *p);`

`FTextToPath` creates a path from a string (at the address **text**) that represents a path in the platform-specific syntax for a path.

Parameter	Type	Description
text	`UPtr`	String that contains the path in platform-specific syntax.
tlen	`int32`	Number of characters in **text**.
p	`Path *`	Address at which `FTextToPath` stores the resulting path. If the value at **p** is non-zero, the function assumes that it is a valid path, resizes the path, and fills in its value. If the value at **p** is zero (NULL), the function creates a new path, fills in its value, and stores the path at the address referred to by **p**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mFullErr`	Insufficient memory.

FUnFlattenPath

syntax `int32 FUnFlattenPath(fp, pPtr);`

FUnFlattenPath converts a flattened path (created using FFlattenPath) into a path.

Parameter	Type	Description
fp	UPtr	Pointer to the flattened path you want to convert to a path.
pPtr	Path *	Address at which FUnFlattenPath stores the resulting path. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns The number of bytes the function interpreted as a path.

Comparing Paths

FIsAPath

syntax `Bool32 FIsAPath(path);`

FIsAPath determines whether **path** is a valid path.

Parameter	Type	Description
path	Path	Path whose validity you want to determine.

returns A boolean, which can have the following values for this function.

Value	Description
TRUE	Path is well formed and type is absolute or relative.
FALSE	Otherwise.

FIsAPathOrNotAPath

syntax `Bool32 FIsAPathOrNotAPath(path);`

FIsAPathOrNotAPath determines whether **path** is a valid path or the canonical invalid path.

Parameter	Type	Description
path	Path	Path whose validity you want to determine.

returns A boolean, which can have the following values for this function.

Value	Description
TRUE	Path is well formed, and type is absolute, relative, or not a path.
FALSE	Otherwise.

FIsEmptyPath

syntax `Bool32 FIsEmptyPath(path);`

`FIsEmptyPath` determines whether **path** is a valid empty path.

Parameter	Type	Description
path	`Path`	Path whose validity and emptiness you want to determine.

returns A boolean, which can have the following values for this function.

Value	Description
TRUE	Path is well formed and empty, and type is absolute or relative.
FALSE	Otherwise.

FPathCmp

syntax `int32 FPathCmp(lsp1, lsp2);`

`FPathCmp` compares the two specified paths.

Parameter	Type	Description
lsp1	`Path`	First path to compare.
lsp2	`Path`	Second path to compare.

returns `int32`, which can have the following values for this function.

Value	Description
-1	Paths are of different types (for example, one is absolute and the other is relative).
0	Paths are identical.
n+1	Paths have the same first *n* components, but are not identical.

Determining a Path Type

FGetPathType

syntax `MgErr FGetPathType(path, typePtr)`

`FGetPathType` returns the type (relative, absolute, or not a path) of the specified path.

Parameter	Type	Description
path	`Path`	Path whose type you want to determine.
typePtr	`int32 *`	Address at which `FGetPathType` stores the type. ***typePtr** can have the following values: • `fAbsPath`: The path is an absolute path. • `fRelPath`: The path is a relative path. • `fNotAPath`: The path is the canonical invalid path or an error occurred. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.

FIsAPathOfType

syntax `Bool32 FIsAPathOfType(path, ofType);`

`FIsAPathOfType` determines whether the specified path is a valid path of the specified type (relative or absolute).

Parameter	Type	Description
path	`Path`	Path that you want to compare to the specified type.
ofType	`int32`	Type that you want to compare to the path's type. **type** can have the following values: • `fAbsPath`: Compare the path's type to absolute. • `fRelPath`: Compare the path's type to relative.

returns A boolean, which can have the following values for this function.

Values	Description
TRUE	Path is well formed and type is identical to **ofType**.
FALSE	Otherwise.

FSetPathType

syntax `MgErr FSetPathType(path, type);`

`FSetPathType` changes the type of the specified path (which must be a valid path) to the specified type (relative or absolute).

Parameter	Type	Description
path	`Path`	Path whose type you want to change.
type	`int32`	New type that you want the path to have. **type** can have the following values: • `fAbsPath`: The path is an absolute path. • `fRelPath`: The path is a relative path.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	Badly formed path or invalid type.

Manipulating File Refnums

FDisposeRefNum

syntax `MgErr FDisposeRefNum(refNum);`

`FDisposeRefNum` disposes of the specified file refnum.

Parameter	Type	Description
refNum	`LVRefNum`	File refnum of which you want to dispose.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	Invalid file refnum.

FIsARefNum

syntax `Bool32 FIsARefNum(refNum);`

`FIsARefNum` determines whether **refNum** is a valid file refnum.

Parameter	Type	Description
refNum	`LVRefNum`	File refnum whose validity you want to determine.

returns A boolean, which can have the following values for this function.

Value	Description
`TRUE`	File refnum has been created and not yet disposed.
`FALSE`	Otherwise.

FNewRefNum

syntax `MgErr FNewRefNum(path, fd, refNumPtr);`

`FNewRefNum` creates a new file refnum for an open file with the name and location specified by **path** and the file descriptor **fd**.

If the file refnum is created, the resulting file refnum is stored in the address referred to by **refNumPtr**. If an error occurs, `NULL` is stored in the address referred to by **refNumPtr** and the error is returned.

Parameter	Type	Description
path	`Path`	The path of the open file for which you wish to create a file refnum.
fd	`File`	The file descriptor of the open file for which you wish to create a file refnum.
refNumPtr	`LVRefNum *`	Address at which `FNewRefNum` stores the new file refnum. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.

FRefNumToFD

syntax `MgErr FRefNumToFD(refNum, fdp);`

`FRefNumToFD` gets the file descriptor associated with the specified file refnum.

If no error occurs, the resulting file descriptor is stored in the address referred to by **fdp**. If an error occurs, `NULL` is stored in the address referred to by **fdp** and the error is returned.

Parameter	Type	Description
refNum	`LVRefNum`	The file refnum whose associated file descriptor you wish to get.
fdp	`File *`	Address at which `FRefNumToFD` stores the file descriptor associated with the specified file refnum. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
`mgArgErr`	Invalid file refnum.

FRefNumToPath

syntax `MgErr FRefNumToPath(refNum, path);`

`FRefNumToPath` gets the path associated with the specified file refnum, and stores the resulting path in the existing path, **path**.

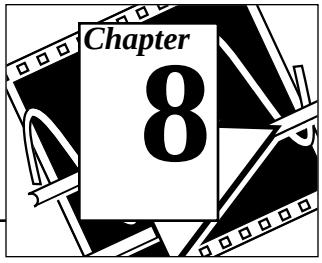
If no error occurs, **path** is set to the path associated with the specified file refnum. If an error occurs, **path** is set to the canonical invalid path.

Parameter	Type	Description
refNum	`LVRefNum`	The file refnum whose associated path you wish to get.
path	`Path`	Path where `FRefNumToPath` stores the path associated with the specified file refnum. This path must already have been created.

returns `MgErr`, which can contain the errors in the following list.

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
mFullErr	Insufficient memory.

Support Manager Functions



Byte Manipulation Operations

Cat4Chrs *Macro*

syntax `int32 Cat4Chrs(a,b,c,d);`

`Cat4Chrs` constructs an `int32` from four `uInt8`s, with the first parameter as the high byte and the last parameter as the low byte.

Parameter	Type	Description
a	`uInt8`	High order byte of the high word of the resulting `int32`.
b	`uInt8`	Low order byte of the high word of the resulting `int32`.
c	`uInt8`	High order byte of the low word of the resulting `int32`.
d	`uInt8`	Low order byte of the low word of the resulting `int32`.

returns The resulting `int32`.

GetALong *Macro*

syntax `int32 GetALong(p);`

`GetALong` retrieves an `int32` from a `void` pointer. On the SPARCstation, this function can retrieve an `int32` at any address, even if the `int32` is not long word aligned.

Parameter	Type	Description
p	`void *`	Address from which you wish to read an `int32`.

returns `int32` stored at the specified address.

Hi16 *Macro*

syntax `int16 Hi16(x);`

`Hi16` returns the high order `int16` of an `int32`.

Parameter	Type	Description
x	`int32`	`int32` of which you want to determine the high `int16`.

HiByte *Macro*

syntax `int8 HiByte(x);`

`HiByte` returns the high order `int8` of an `int16`.

Parameter	Type	Description
x	`int16`	`int16` of which you want to determine the high `int8`.

HiNibble *Macro*

syntax `uInt8 HiNibble(x);`

`HiNibble` returns the value stored in the high four bits of an `uInt8`.

Parameter	Type	Description
x	`uInt8`	`uInt8` whose high four bits you want to extract.

Lo16 *Macro*

syntax `int16 Lo16(x);`

`Lo16` returns the low order `int16` of an `int32`.

Parameter	Type	Description
x	`int32`	`int32` of which you want to determine the low `int16`.

HiNibble *Macro*

syntax `uInt8 HiNibble(x);`

`HiNibble` returns the value stored in the high four bits of an `uInt8`.

Parameter	Type	Description
x	`uInt8`	`uInt8` whose high four bits you want to extract.

LoByte *Macro*

syntax `int8 LoByte(x);`

`LoByte` returns the low order `int8` of an `int16`.

Parameter	Type	Description
x	`int16`	`int16` of which you want to determine the low `int8`.

Long *Macro*

syntax `int32 Long(hi, lo);`

`Long` creates an `int32` from two `int16`s.

Parameter	Type	Description
hi	`int16`	High `int16` for the resulting `int32`.
lo	`int16`	Low `int16` for the resulting `int32`.

returns The resulting `int32`.

LoNibble *Macro*

syntax `uInt8 LoNibble(x);`

`LoNibble` returns the value stored in the low four bits of an `uInt8`.

Parameter	Type	Description
x	`uInt8`	`uInt8` whose low four bits you want to extract.

Offset *Macro*

syntax `int16 Offset(type, field);`

`Offset` returns the offset of the specified field within the structure called **type**.

Parameter	Type	Description
type	-	Structure that contains field.
field	-	Field whose offset you want to determine.

returns An offset as an `int16`.

SetALong *Macro*

syntax `void SetALong(p,x);`

`SetALong` stores an `int32` at the address specified by a `void` pointer. On the SPARCstation, this function can retrieve an `int32` at any address, even if it is not long word aligned.

Parameter	Type	Description
p	`void *`	Address at which you want to store an `int32`. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.
x	`int32`	Value that you want to store at the specified address.

Word *Macro*

syntax `int16 Word(hi, lo);`

`Word` creates an `int16` from two `int8`s.

Parameter	Type	Description
hi	`int8`	High `int8` for the resulting `int16`.
lo	`int8`	Low `int8` for the resulting `int16`.

returns The resulting `int16`.

Mathematical Operations

In addition to the mathematical operations documented in this section, LabVIEW supports a number of other mathematical functions. These functions are implemented as defined in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. Table 8.1 lists the prototypes for these functions.

Table 8-1. Mathematical Functions Supported by LabVIEW

`double atan(double);`
`double cos(double);`
`double exp(double);`
`double fabs(double);`
`double log(double);`
`double sin(double);`
`double sqrt(double);`
`double tan(double);`
`double acos(double);`
`double asin(double);`
`double atan2(double, double);`
`double ceil(double);`
`double cosh(double);`
`double floor(double);`
`double fmod(double, double);`
`double frexp(double, int *);`
`double ldexp(double, int);`
`double log10(double);`
`double modf(double, double *);`
`double pow(double, double);`
`double sinh(double);`
`double tanh(double);`

For THINK C Users

To link the math functions when using THINK C, you need to add additional files to your project. You can link a modified version of an ANSI library provided by THINK C. The ANSI library must be modified to reference its globals from A4 instead of A5; this process is explained in the THINK C documentation in the section concerning building code resources (the section has different names in the various THINK C versions).

To make such a library, make a copy of the `ANSI-A4` project (shipped with THINK C), and name it `ANSI-A4 copy` (or any unique name). Add the `math.c` file (shipped with THINK C) to `ANSI-A4 copy`, and then select **Build Library...** under the **Project** menu. Name your new library `mathlib` (or any unique name). Adding `mathlib` to your CIN project makes it possible for your math functions to link.

Abs

syntax `int32 Abs(n);`

`Abs` returns the absolute value of **n**, unless **n** is -2^31, in which case the function returns the number unmodified.

Parameter	Type	Description
n	`int32`	`int32` whose absolute value you want to find.

Max

syntax `int32 Max(n,m);`

`Max` returns the maximum of the two specified `int32`s.

Parameter	Type	Description
n,m	`int32`	`int32`s whose maximum value you want to determine.

Min

syntax `int32 Min(n,m);`

`Min` returns the minimum of the two specified `int32`s.

Parameter	Type	Description

n,m	`int32`	`int32`s whose minimum value you want to determine.

Pin

syntax `int32 Pin(i,low,high);`

`Pin` returns **i** coerced to fall within the range from **low** to **high** inclusive.

Parameter	Type	Description
i	`int32`	Value you want to coerce to the specified range.
n	`int32`	Low value of the range to which you want to coerce **i**.
m	`int32`	High value of the range to which you want to coerce **i**.

returns i coerced to the specified range.

RandomGen

syntax `void RandomGen(xp);`

`RandomGen` generates a random number between 0 and 1 and stores it at **xp**.

Parameter	Type	Description
xp	`float64 *`	Location to store the resulting double-precision floating-point random number. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

String Manipulation

BlockCmp

syntax `int32 BlockCmp(p1, p2, numBytes);`

`BlockCmp` compares two blocks of memory to determine whether one is less than, the same as, or greater than the other.

Parameter	Type	Description
p1	`UPtr`	Pointer to a block of memory.
p2	`UPtr`	Pointer to a block of memory.
numBytes	`int32`	Number of bytes to compare.

returns A negative number, zero, or a positive number if **s1** is less than, the same as, or greater than **s2**.

CPStrBuf *Macro*

syntax `uChar *CPStrBuf(sp);`

`CPStrBuf` returns the address of the first string in a concatenated list of Pascal strings (that is, the address of `sp->str`).

Parameter	Type	Description
sp	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.

returns The address of the first string of the concatenated list of Pascal strings.

CPStrCmp

syntax `int32 CPStrCmp(s1p, s2p);`

`CPStrCmp` lexically compares two concatenated lists of Pascal strings to determine whether one is less than, the same as, or greater than the other. This comparison is case sensitive, and the function compares the lists as if they were one string.

Parameter	Type	Description
s1p	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.
s2p	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2.**

CPStrIndex

syntax `PStr CPStrIndex(s1h, index);`

`CPStrIndex` returns a pointer to the Pascal string denoted by **index** in a list of strings. If **index** is greater than or equal to the number of strings in the list, the function returns the pointer to the last string.

Parameter	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
index	`int32`	Number of the string that you want, with 0 as the first string.

returns A pointer to the specified Pascal string.

CPStrInsert

syntax

```
MgErr       CPStrInsert(s1h, s2, index);
```

`CPStrInsert` inserts a new Pascal string before the **index** numbered Pascal string in a concatenated list of Pascal strings. If **index** is greater than or equal to the number of strings in the list, the function places the new string at the end of the list. `CPStrInsert` resizes the list to make room for the new string.

Parameter	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
s2	`PStr`	Pointer to a Pascal string.
index	`int32`	Position that you want the new Pascal string to have in the list of Pascal strings, with 0 as the first string.

returns `mFullErr` if there is not enough memory. Returns `noErr` otherwise.

CPStrLen *Macro*

syntax

```
int32       CPStrLen(sp);
```

`CPStrLen` returns the number of Pascal strings in a concatenated list of Pascal strings (that is, `sp->cnt`). Use the `CPStrSize` function to get the total number of characters in the list.

Parameter	Type	Description
sp	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.

returns The number of strings in the concatenated list of Pascal strings.

CPStrRemove

syntax `void CPStrRemove(s1h, index);`

`CPStrRemove` removes a Pascal string from a list of Pascal strings. If **index** is greater than or equal to the number of strings in the list, the function removes the last string. `CPStrRemove` resizes the list after removing the string.

Parameter	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
index	`int32`	Number of the string that you want to remove, with 0 as the first string.

CPStrReplace

syntax `MgErr CPStrReplace(s1h, s2, index);`

`CPStrReplace` replaces a Pascal string in a concatenated list of Pascal strings with a new Pascal string.

Parameter	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
s2	`PStr`	Pointer to a Pascal string.
index	`int32`	Number of the string that you want to replace, with 0 as the first string.

returns `mFullErr` if there is not enough memory. Returns `noErr` otherwise.

CPStrSize

syntax `int32 CPStrSize(sp);`

`CPStrSize` returns the number of characters in a concatenated list of Pascal strings. Use the `CPStrLen` function to get the number of Pascal strings in the concatenated list.

Parameter	Type	Description
sp	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.

returns The number of characters in the concatenated list of Pascal strings.

CToPStr

syntax `int32 CToPStr(cstr, pstr);`

`CToPStr` converts a C string to a Pascal string. This function works even if the pointers **cstr** and **pstr** refer to the same memory location. If the length of **cstr** is greater than 255 characters, the function converts only the first 255 characters. The function assumes that **pstr** is large enough to contain **cstr**.

Parameter	Type	Description
cstr	`CStr`	Pointer to a C string.
pstr	`PStr`	Pointer to a Pascal string.

returns The length of the string, truncated to a maximum of 255 characters.

FileNameCmp *Macro*

syntax `int32 FileNameCmp(s1, s2);`

`FileNameCmp` lexically compares two file names, to determine whether one is less than, the same as, or greater than the other. This comparison uses the same case sensitivity as the file system (that is, case insensitive for the Macintosh and the PC, case sensitive for the Sun SPARCstation).

Parameter	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2**.

FileNameIndCmp *Macro*

syntax `int32 FileNameIndCmp(s1p, s2p);`

`FileNameIndCmp` is the same as `FileNameCmp`, except you pass the function handles to the string data instead of pointers. You can use `FileNameIndCmp` to compare two file names and lexically determine whether one is less than, the same as, or greater than the other. This comparison uses the same case sensitivity as the file

system (that is, case insensitive for the Macintosh and the PC, and case sensitive for the Sun SPARCstation).

Parameter	Type	Description
s1p	`PStr *`	Pointer to a Pascal string.
s2p	`PStr *`	Pointer to a Pascal string.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2**.

FileNameNCmp *Macro*

syntax `int32 FileNameNCmp(s1, s2, n);`

`FileNameNCmp` lexically compares two file names to determine whether one is less than, the same as, or greater than the other, limiting the comparison to **n** characters. This comparison uses the same case sensitivity as the file system (that is, case insensitive for the Macintosh and the PC, case sensitive for the Sun SPARCstation).

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.
n	`uInt32`	Maximum number of characters to compare.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2**.

HexChar

syntax `int32 HexChar(n);`

`HexChar` returns the ASCII character in hex that represents the specified value **n** (0<=**n**<=15).

Parameter	Type	Description
n	`int32`	Decimal value between 0 and 15.

returns The corresponding ASCII hex character. If **n** is out of range, the ASCII character corresponding to **n** modulo 16 is returned.

IsAlpha

syntax `Bool32 IsAlpha(c);`

`IsAlpha` returns `TRUE` if the character **c** is a lowercase or uppercase letter (that is, in the set a to z or A to Z). On the SPARCstation, this function also returns `TRUE` for international characters (à, á, Ä, and so on).

Parameter	Type	Description
c	`char`	Character that you want to analyze.

returns `TRUE` if the character is an alphabetic character, and `FALSE` otherwise.

IsDigit

syntax `Bool32 IsDigit(c);`

`IsDigit` returns `TRUE` if the character **c** is between 0 and 9.

Parameter	Type	Description
c	`char`	Character that you want to analyze.

returns `TRUE` if the character is a numerical digit, and `FALSE` otherwise.

IsLower

syntax `Bool32 IsLower(c);`

`IsLower` returns `TRUE` if the character **c** is a lowercase letter (that is, in the set a to z). On the SPARCstation, this function also returns `TRUE` for lowercase international characters (ó, ö, and so on).

Parameter	Type	Description
c	`char`	Character that you want to analyze.

returns `TRUE` if the character is a lowercase letter, and `FALSE` otherwise.

IsUpper

syntax `Bool32 IsUpper(c);`

IsUpper returns TRUE if the character **c** is between an uppercase letter (that is, in the set A to Z). On the SPARCstation, this function also returns TRUE for uppercase international characters (Ó, Ä, and so on).

Parameter	Type	Description
c	char	Character that you want to analyze.

returns TRUE if the character is an uppercase letter, and FALSE otherwise.

LStrBuf *Macro*

syntax `uChar *LStrBuf(s);`

LStrBuf returns the address of the string data of a long Pascal string (that is, the address of s->str).

Parameter	Type	Description
s	LStrPtr	Pointer to a long Pascal string.

returns The address of the string data of the long Pascal string.

LStrCmp

syntax `LStrPtr LStrCmp(l1p, l2p);`

LStrCmp lexically compares two long Pascal strings to determine whether one is less than, the same as, or greater than the other. This comparison is case sensitive.

Parameter	Type	Description
l1p	LStrPtr	Pointer to a long Pascal string.
l2p	LStrPtr	Pointer to a long Pascal string.

returns <0, 0, or >0 if **s1** is less than, the same as, or greater than **s2**. Returns <0 if **s1** is an initial substring of **s2.**

LStrLen *Macro*

syntax `int32 LStrLen(s);`

LStrLen returns the length of a long Pascal string (that is, s->cnt).

Parameter	Type	Description
s	`LStrPtr`	Pointer to a long Pascal string.

returns The number of characters in the long Pascal string.

LToPStr

syntax `int32 LToPStr(lstrp, pstr);`

`LToPStr` converts a long Pascal string to a Pascal string. If the long Pascal string is more than 255 characters, the function converts only the first 255 characters. This function works even if the pointers **lstrp** and **pstr** refer to the same memory location. The function assumes that **pstr** is large enough to contain **lstrp**.

Parameter	Type	Description
lstrp	`LStrPtr`	Pointer to a long Pascal string.
pstr	`PStr`	Pointer to a Pascal string.

returns The length of the string, truncated to a maximum of 255 characters.

PPStrCaseCmp

syntax `int32 PPStrCaseCmp(s1p, s2p);`

`PPStrCaseCmp` is the same as `PStrCaseCmp`, except you pass the function handles to the string data instead of pointers. You can use `PPStrCaseCmp` to compare two Pascal strings lexically and determine whether one is less than, the same as, or greater than the other. This comparison ignores differences in case.

Parameter	Type	Description
s1p	`PStr *`	Pointer to a Pascal string.
s2p	`PStr *`	Pointer to a Pascal string.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2.**

PPStrCmp

syntax `int32 PPStrCmp(s1p, s2p);`

PPStrCmp is the same as PStrCmp, except you pass the function handles to the string data instead of pointers. You can use PPStrCmp to compare two Pascal strings lexically and determine whether one is less than, the same as, or greater than the other. This comparison is case sensitive.

Parameter	Type	Description
s1p	PStr *	Pointer to a Pascal string.
s2p	PStr *	Pointer to a Pascal string.

returns <0, 0, or >0 if **s1** is less than, the same as, or greater than **s2**. Returns <0 if **s1** is an initial substring of **s2.**

PStrBuf *Macro*

syntax `uChar *PStrBuf(s);`

PStrBuf returns the address of the string data of a Pascal string (that is, the address following the length byte).

Parameter	Type	Description
s	PStr	Pointer to a Pascal string.

PStrCaseCmp

syntax `int32 PStrCaseCmp(s1, s2);`

PStrCaseCmp lexically compares two Pascal strings to determine whether one is less than, the same as, or greater than the other. This comparison ignores differences in case.

Parameter	Type	Description
s1	PStr	Pointer to a Pascal string.
s2	PStr	Pointer to a Pascal string.

returns <0, 0, or >0 if **s1** is less than, the same as, or greater than **s2**. Returns <0 if **s1** is an initial substring of **s2.**

PStrCat

syntax `int32`

```
PStrCat(s1, s2);
```

`PStrCat` concatenates a Pascal string, **s2**, to the end of another Pascal string, **s1**, and places the result in **s1**. This function assumes that **s1** is large enough to contain the resulting string. If the resulting string is larger than 255 characters, then `PStrCat` limits the resulting string to 255 characters.

Parameter	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

returns The length of the resulting string.

PStrCmp

syntax `int32 PStrCmp(s1, s2);`

`PStrCmp` lexically compares two Pascal strings to determine whether one is less than, the same as, or greater than the other. This comparison is case sensitive.

Parameter	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2.**

PStrCpy

syntax `PStr PStrCpy(dst, src);`

`PStrCpy` copies the Pascal string **src** to the Pascal string **dst**. This function assumes that the destination string is large enough to contain the source string.

Parameter	Type	Description
dst	`PStr`	Pointer to a Pascal string.
src	`PStr`	Pointer to a Pascal string.

returns **A copy of the destination Pascal string pointer.**

PStrLen *Macro*

syntax `uInt8 PStrLen(s);`

`PStrLen` returns the length of a Pascal string (that is, the value at the first byte at the specified address).

Parameter	Type	Description
s	`PStr`	Pointer to a Pascal string.

PStrNCpy

syntax `PStr PStrNCpy(dst, src, n);`

`PStrNCpy` copies the Pascal string **src** to the Pascal string **dst**. If the source string is greater than **n**, the function copies only **n** bytes. This function assumes that the destination string is large enough to contain the source string.

Parameter	Type	Description
dst	`PStr`	Pointer to a Pascal string.
src	`PStr`	Pointer to a Pascal string.
n	`int32`	Maximum number of bytes to copy including the length byte.

returns A copy of the destination Pascal string pointer.

PToCStr

syntax `int32 PToCStr(pstr, cstr);`

`PToCStr` converts a Pascal string to a C string. This function works even if the pointers **pstr** and **cstr** refer to the same memory location. This function assumes that **cstr** is large enough to contain **pstr**.

Parameter	Type	Description
pstr	`PStr`	Pointer to a Pascal string.
cstr	`CStr`	Pointer to a C string.

returns The length of the string.

PToLStr

syntax

```
int32       PToLStr(pstr, lstrp);
```

`PToLStr` converts a Pascal string to a long Pascal string. This function works even if the pointers **pstr** and **lstrp** refer to the same memory location. The function assumes that **lstrp** is large enough to contain **pstr**.

Parameter	Type	Description
pstr	`PStr`	Pointer to a Pascal string.
lstrp	`LStrPtr`	Pointer to a long Pascal string.

returns The length of the string.

SPrintf
SPrintfp
PPrintf
PPrintfp
FPrintf
LStrPrintf

syntax

```
int32       SPrintf(CStr destCSt, CStr cfmt, ...);
int32       SPrintfp(CStr destCSt, PStr pfmt, ...);
int32       PPrintf(PStr destPSt, CStr cfmt, ...);
int32       PPrintfp(PStr destPSt, PStr pfmt, ...);
int32       FPrintf(File destFile, CStr cfmt, ...);
MgErr       LStrPrintf(LStrHandle destLsh, CStr cfmt,
            ...);
```

All these functions format data into an ASCII format to a specified destination. A format string describes the desired conversions. These functions take a variable number of arguments, and each argument follows the format string paired with a conversion specification embedded in the format string. The second parameter, **cfmt** or **pfmt**, must be cast appropriately to either type `CStr` or `PStr`.

`SPrintf` prints to a C string, just like the C library function `sprintf`. `sprintf` returns the actual character count and appends a null byte to the end of the destination C string.

`SPrintfp` is the same as `SPrintf`, except the format string is a Pascal string instead of a C string. As with `SPrintf`, `SPrintfp` appends a null byte to the end of the destination C string.

If you pass NULL for **destCStr**, SPrintf and SPrintfp do not write data to memory, and they return the number of characters required to contain the resulting data (not including the terminating null character).

PPrintf prints to a Pascal string with a maximum of 255 characters. PPrintf sets the length byte of the Pascal string to reflect the size of the resulting string. PPrintf does not append a null byte to the end of the string.

PPrintfp is the same as PPrintf, except the format string is a Pascal string instead of a C string. As with PPrintf, PPrintfp sets the length byte of the Pascal string to reflect the size of the resulting string.

FPrintf prints to a file specified by the refnum in **fd**. FPrintf does not embed a length count or a terminating null character in the data written to the file.

LStrPrintf prints to a LabVIEW string specified by destLsh. Because the LabVIEW string is a handle that may be resized, LStrPrintf can return memory errors just as DSSetHandleSize does.

These functions accept the following standard formats and special characters.

- Special characters that can be embedded in strings:
  - \b backspace
  - \f form feed
  - \n new line (inserts the system-dependent end-of-line char(s); for example, CR on Macintosh, NL on UNIX, CRNL on DOS)
  - \r carriage return
  - \s space
  - \t tab
  - %% percentage character (to print %)
- Format arguments:

```
%[-] [field size] [.precision] [argument size] [conversion]
```

  - [-] Left-justifies what is printed; if not specified, the data is right-justified.
  - [field size] Specifies the minimum width of the field to print into. If not specified, this defaults to 0. If there is less than the specified number of characters in the data to print, the function pads with spaces on the left if you specified -; otherwise the function pads on the right.
  - [.precision] Sets the precision for floating-point numbers (that is, the number of characters after the decimal place). For strings, this specifies the maximum number of characters to print.

- `[argument size]` Specifies the data size for an argument. It applies only to the **d**, **o**, **u**, and **x** conversion specifiers. By default, the conversion for one of the specifiers is from a word (16-bit integer). The flag **l** causes this conversion to convert the data so that the function assumes the data is a long integer value.
- `[conversion]` `b` binary

  `c` print a character (%2c, %4c print on int16, int32 as a 2,4 char constant)

  `d` decimal

  `e` exponential

  `f` fixed point format

  `H` string handle (LStrHandle)

  `o` octal

  `p` Pascal string

  `P` long Pascal string (LStrPtr)

  `q` print a point (passed by value) as %d,%d representing horizontal, vertical coordinates

  `Q` print a point (passed by value) as hv(%d,%d) representing horizontal, vertical coordinates

  `r` print a rectangle (passed by reference) as %d,%d,%d,%d representing top,left, bottom, right coordinates

  `R` print a rectangle (passed by reference) as tlbr(%d,%d,%d,%d) representing top,left, bottom, right coordinates

  `s` string

  `u` unsigned decimal

  `x` hex

  `z` Path

Any of the numeric conversion characters (`x`, `o`, `d`, `u`, `b`, `e`, `f`) can be preceded by {`cc`} to indicate that the number is passed by reference. `cc` can be `iB`, `iW`, … , `cX` depending on the corresponding numeric type. If `cc` is an asterisk (`*`) the numeric type (`iB` through `cX`) is an `int16` in the argument list.

StrCat

syntax `int32 StrCat(s1, s2);`

`StrCat` concatenates a C string, **s2**, to the end of another C string, **s1**, placing the result in **s1**. This function assumes that **s1** is large enough to contain the resulting string.

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.

returns The length of the resulting string.

StrCmp

syntax `int32 StrCmp(s1, s2);`

`StrCmp` lexically compares two strings to determine whether one is less than, the same as, or greater than the other.

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2**.

StrCpy

syntax `CStr StrCpy(dst, src);`

`StrCpy` copies the C string **src** to the C string **dst**. This function assumes that the destination string is large enough to contain the source string.

Parameter	Type	Description
dst	`CStr`	Pointer to a C string.
src	`CStr`	Pointer to a C string.

returns A copy of the destination C string pointer.

StrLen

syntax `int32 StrLen(s);`

`StrLen` returns the length of a C string.

Parameter	Type	Description
s	`CStr`	Pointer to a C string.

returns The number of characters in the C string, not including the `NULL` terminating character.

StrNCaseCmp

syntax `int32 StrNCaseCmp(s1, s2, n);`

`StrNCaseCmp` lexically compares two strings to determine whether one is less than, the same as, or greater than the other, limiting the comparison to **n** characters. `StrNCaseCmp` ignores differences in case in performing the comparison.

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.
n	`uInt32`	Maximum number of characters to compare.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2**.

StrNCmp

syntax `int32 StrNCmp(s1, s2, n);`

`StrNCmp` lexically compares two strings to determine whether one is less than, the same as, or greater than the other, limiting the comparison to **n** characters.

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.
n	`uInt32`	Maximum number of characters to compare.

returns `<0`, `0`, or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2**.

StrNCpy

syntax `CStr StrNCpy(dst, src, n);`

`StrNCpy` copies the C string **src** to the C string **dst**. If the source string is less than **n** characters, the function pads the destination with null characters. If the source string is greater than **n**, then only **n** characters are copied. This function assumes that the destination string is large enough to contain the source string.

Parameter	Type	Description
dst	`CStr`	Pointer to a C string.
src	`CStr`	Pointer to a C string.
n	`int32`	Maximum number of characters to copy.

returns A copy of the destination C string pointer.

ToLower

syntax `uChar ToLower(c);`

`ToLower` returns the lowercase value of **c** if **c** is an uppercase alphabetic character. Otherwise, it returns **c** unmodified. On the SPARCstation, this function also works for international characters (Ä -> ä, and so on).

Parameter	Type	Description
c	`uChar`	Character that you want to analyze.

returns The lowercase value of **c**.

ToUpper

syntax `uChar ToUpper(c);`

`ToUpper` returns the uppercase value of **c** if **c** is a lowercase alphabetic character. Otherwise, it returns **c** unmodified. On the SPARCstation, this function also works for international characters (ä -> Ä, and so on).

Parameter	Type	Description
c	`uChar`	Character that you want to analyze.

returns The uppercase value of **c**

Utility Functions

BinSearch

syntax `int32 BinSearch(arrayp, n, elmtSize, key, compareProcP);`

`BinSearch` searches an array of an arbitrary data type using the binary search algorithm. In addition to passing the array that you want to search to this routine, you also pass a comparison procedure that this sort routine then uses to compare elements in the array.

The comparison routine should return a number less than zero if **a** is less than **b**, zero if **a** is equal to **b**, and a number greater than zero if **a** is greater than **b**.

You should declare the comparison routine to have the following parameters and return type.

int32 compareProcP(UPtr a, UPtr b);

Parameter	Type	Description
arrayp	`UPtr`	Pointer to an array of data.
n	`int32`	Number of elements in the array that you want to search.
elmtSize	`int32`	Size in bytes of an array element.
key	`UPtr`	Pointer to the data that you want to search for.
compareProcP	`procPtr`	Comparison routine that you want `BinSearch` to use in comparing array elements. `BinSearch` passes this routine the addresses of two elements that it needs to compare.

returns The position in the array where the data is found (with `0` being the first element of the array), if it is found. If the data is not found, BinSearch returns `-i-1`, where `i` is the position where `x` should be placed.

QSort

syntax `void QSort(arrayp,n,elmtSize, compareProcP());`

`QSort` sorts an array of an arbitrary data type using the QuickSort algorithm. In addition to passing the array that you want to sort to this routine, you also pass a comparison procedure that this sort routine then uses to compare elements in the array.

The comparison routine should return a number less than zero if **a** is less than **b**, zero if **a** is equal to **b**, and a number greater than zero if **a** is greater than **b**.

You should declare the comparison routine to have the following parameters and return type.

int32 compareProcP(UPtr a, UPtr b);

Parameter	Type	Description
arrayp	`UPtr`	Pointer to an array of data.
n	`int32`	Number of elements in the array that you want to sort.
elmtSize	`int32`	Size in bytes of an array element.
compareProcP	`procPtr`	Comparison routine that you want `QSort` to use to compare array elements. `QSort` passes this routine the addresses of two elements that it needs to compare.

Unused *Macro*

syntax `void Unused(x)`

`Unused` indicates that a function parameter or local variable is not used by that function. This is useful for suppressing compiler warnings for many compilers. Notice that no semicolon is used with this macro.

Parameter	Type	Description
x	-	Unused parameter or local variable.

Time Functions

ASCIITime

syntax `CStr ASCIITime(secs);`

`ASCIITime` returns a pointer to a string representing the date and time of day corresponding to *t* seconds after January 1, 1904, 12:00 AM, GMT. This function uses the same date format as that returned by the `DateCString` function using a mode of 2. The date is followed by a space, and the time is in the same format as that returned by the `TimeCString` function using a mode of 0. As an example, this function might return Tuesday, Dec 22, 1992 5:30. On the SPARCstation, this function accounts for international conventions for representing dates.

Parameter	Type	Description
secs	`uInt32`	Seconds since the January 1, 1904, 12:00 AM, GMT.

returns The date and time as a C string.

DateCString

syntax `CStr DateCString(secs, fmt);`

Note: ***This function was formerly called*** `DateString`**.**

`DateCString` returns a pointer to a string representing the date corresponding to *t* seconds after January 1, 1904, 12:00 AM, GMT. On the SPARCstation, this function accounts for international conventions for representing dates.

Parameter	Type	Description
secs	`uInt32`	Seconds since January 1, 1904, 12:00 AM, GMT.
fmt	`int32`	Code describing the format for the returned string. This parameter determines the format of the returned date string and can have the following values.

Fmt	Meaning
0	Return the date in short date format, *mm*/*dd*/*yy*, where *mm* is a number between 1 and 12 representing the current month, *dd* is the current day of the month (1 through

31), and *yy* is the last two digits of the corresponding year. An example is 12/31/92.

1 Return the date in long date format, *dayName*, *MonthName*, *DayOfMonth*, *LongYear*. An example is Thursday, December 31, 1992.

2 Return the date in abbreviated date format, *AbbrevDayName*, *AbbrevMonthName*, *DayOfMonth*, *LongYear*. An example is Thu, Dec 31, 1992.

returns The date as a C string.

DateToSecs

syntax `uInt32 DateToSecs(dateRecordP);`

`DateToSecs` converts from a time described using the `DateRec` data structure to the number of seconds since January 1, 1904, 12:00 AM, GMT.

Parameter	Type	Description
dateRecordP	`DateRec *`	Pointer to a `DateRec` structure. `DateToSecs` stores the converted date in the fields of the date structure referred to by **dateRecordP**. This data structure is described in the *Introduction* section of this chapter. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

returns The corresponding number of seconds since January 1, 1904, 12:00 AM, GMT.

MilliSecs

syntax `uInt32 MilliSecs();`

returns The time since an undefined system time in milliseconds. The actual resolution of this timer is system dependent.

SecsToDate

syntax `void SecsToDate(secs, dateRecordP);`

`SecsToDate` converts the seconds since January 1, 1904, 12:00 AM, GMT into a data structure containing numerical information about the date, including the year (1904 through 2040), the month (1 through 12), the day as it corresponds to the current year (1 through 366), month (1 through 31), and week (1 through 31), hour (0 through 23), the hour (0 through 23), minute (0 through 59), and second (0 through 59) of that day, and a value indicating whether the time specified uses daylight savings time.

Parameter	Type	Description
secs	`uInt32`	Seconds since January 1, 1904, 12:00 AM, GMT.
dateRecordP	`DateRec *`	Pointer to a `DateRec` structure. `SecsToDate` stores the converted date in the fields of the date structure referred to by **dateRecordP**. This data structure is described in the *Introduction* section of this chapter. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, for more information about using this parameter.

TimeCString

syntax `CStr TimeCString(secs, fmt);`

Note: ***This function was formerly called*** `TimeString`**.**

`TimeCString` returns a pointer to a string representing the time of day corresponding to *t* seconds after January 1, 1904, 12:00 AM, GMT. On the SPARCstation, this function accounts for international conventions for representing dates.

Parameter	Type	Description
secs	`uInt32`	Seconds since January 1, 1904, 12:00 AM, GMT.
fmt	`int32`	Code describing the format for the returned string. The parameter **fmt** determines the format of the returned time string and can have the following values. **Fmt Meaning**

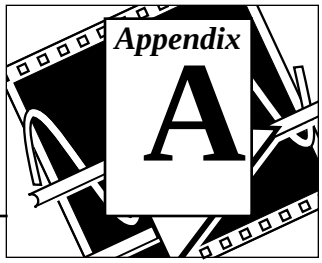
0	Return the time in the format *hh*:*mm*. The first value, *hh*, represents the hour (0 through 23, with 0 as midnight), and the second value, *mm*, represents the minute (0 through 59).
1	Return the time in the format *hh*:*mm*:*ss*. The first value, *hh*, represents the hour, the second value, *mm*, represents the minute (0 through 59), and the third value, *ss*, represents the second (0 through 59).

returns The time as a C string.

TimeInSecs

syntax `uInt32 TimeInSecs();`

returns The current date and time in seconds relative to January 1, 1904, 12:00 AM, Greenwich mean time (GMT).



CIN Common Questions

This appendix answers some of the questions commonly asked by LabVIEW CIN users.

What compilers can be used to write CINs for LabVIEW?

Microsoft Windows 3.1, Windows 95, and Windows NT

You can use the Watcom C/386 compiler, version 9.0 or later, to write CINs for LabVIEW for Windows 3.1. Other compilers for Windows 3.1 (including the Microsoft C compiler) do not generate the proper code for LabVIEW to operate as a 32-bit application. For a compiler to work with LabVIEW, it must generate a file in the `.REX` format (a 32-bit Phar Lap relocatable executable).

LabVIEW for Windows 95/NT supports additional compilers, including Microsoft C/C++ and Visual C++ for NT.

Macintosh

You can use the following compilers to compile your CIN source code: THINK C, version 5-7, for 68K (from Symantec Corporation of Cupertino, CA); Symantec C++, version 8, for PowerPC (from Symantec Corporation of Cupertino, CA); Metrowerks CodeWarrior for 68K (from Metrowerks Corporation of Austin, TX); Metrowerks CodeWarrior for Power Macintosh (from Metrowerks Corporation of Austin, TX); Macintosh Programmer's Workshop (MPW) for 68K and PowerPC (from Apple Computer, Inc. of Cupertino, CA).

Sun

You can use the Sun ANSI-compatible compiler and the `gcc` compiler. The only officially supported compiler is the ANSI C compiler, also known as the unbundled C compiler or SPARCompiler C, which can be purchased from Sun. On Solaris 1.*x* machines, this compiler is commonly referred to as `acc` (ANSI C compiler); on Solaris 2.*x* machines, the compiler is called `cc`. The Gnu C compiler (`gcc`) is also ANSI-compatible and can be used to create CINs for LabVIEW for Sun. The only known limitation of the `gcc` compiler is that, under

Solaris 1.*x*, it does not support extended-precision floating point numbers. Source code for the `gcc` compiler is available for both Solaris 1.*x* and 2.*x* through anonymous ftp to `prep.ai.mit.edu`.

SPARCstations with Solaris 1.*x* come with the bundled C compiler (`cc`) that is not ANSI-compliant. Because the `cc` compiler requires substantial modification to the header files included with LabVIEW, National Instruments does not recommend using this compiler for CIN development.

Please note that LabVIEW for Solaris 1.*x* does not accept object files created with the `-g` debugging flag turned on during compilation.

My VI, which contains a CIN, crashes LabVIEW or gives a memory.c error.

In almost all cases this indicates an error in the C code of the CIN. Make sure that the CIN code properly allocates or deallocates memory as necessary. See the section entitled *How LabVIEW Passes Variably Sized Data to CINs* in Chapter 2, *CIN Parameter Passing*, of this manual for further details and examples.

How do I debug my CIN?

You have several debugging options, depending upon the platform you use. The following list gives descriptions of some of the available methods.

- Use the `DbgPrintf` function, which creates a debugging window. Although the position and size of the window cannot be controlled, information can be posted to the window as the CIN code executes. Notice that the window does not contain a scrollbar. `DbgPrintf` is described in the section entitled *Debugging External Code* in Chapter 1, *CIN Overview*, of this manual.
- If you are using a Macintosh and have `Macsbug`, you can use the `Debugger` and `DebugStr` statements to set breakpoints in the code.
- If you suspect that your CIN is corrupting memory, use `DSHeapCheck(FALSE)` to test for integrity. Observe the heap integrity when you enter and again when you exit the CIN code to determine if your code is corrupting the heap.
- Use the File Manager functions to write your debugging information out to a file. If you are observing this file while the

CIN is running, do not forget to flush the file before the information physically gets to the disk.

- If the VI containing the CIN executes without crashing, but you do not have an external window and decide not to use `DbgPrintf`, then a) determine what information is pertinent to your problem, and b) return the information from one of the parameters of the CIN to the block diagram of the VI.

Is there any sort of scanf function in the LabVIEW manager routines?

No. National Instruments is investigating this functionality for a future version of LabVIEW. CINs with LabVIEW for Sun can call the standard `scanf` and related functions.

I can't seem to link to any of the globals mentioned in the *LabVIEW Code Interface Reference Manual*.

Examples of these globals include: `decimalPt, CrgRtnChar, LnFeedChar, EOLChar, TabChar, EmptyStrChar, SInfinity, SNegInfinity, DInfinity, DNegInfinity, EMaxW, EMaxL, EInfinity, ENegInfinity, DPi, DHalfPi, DThreeHalvesPi, DTwoPi, DRad2Deg, DTwo, DNan, EPi, EHalfPi, ETwoPi, EE, Eln10, Eln2, Elog10e, ELog2e, EHalf, EOne, ETwo, ETen, EZero, ERecipPi, ERecipE, EPlanck, EElemChg, ESpeedLt, EGravity, EAvgdro, ERydbrg, EMlrGas, ELnOfPi, ELogOfE, ELnOfTwo,` and `ENan`.

Although mentioned in the documentation, these globals are not exported for use in CINs. To get these values into your CIN code, pass them in as parameters to the CIN.

Can LabVIEW be used to call a DLL in Windows?

Yes. The new Call Library Function calls a DLL function directly. The function is located in the **Advanced** palette of the **Functions** palette. Refer to Chapter 11 of the *LabVIEW Function Reference Manual* for more details on this new feature.

I get an error linking to a function when I build my CIN using the Windows platform.

The Watcom linker usually does not allow you to link with the Watcom library function modules when making a stand-alone module. If it does allow you to link, the code should work properly. Unfortunately, there is no clearly defined way to determine which functions will link and which will not; it is trial and error.

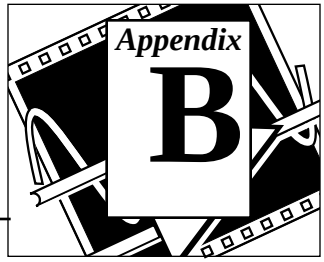
If this error occurs, the only way to work through the problem is to write a DLL that calls the library functions.

Why do I get garbage back from math functions such as atan2, pow, ceil, floor, ldexp, frexp, modf, and fmod when using MPW C?

Include `"Math.h"` at the top of your `.c` file.

Why can't I link to the math functions (sin, cos, and so on) when using THINK C?

Find the `math.c` and `error.c` functions that came with THINK C and include them in the project. Be sure to also include `"Math.h"` in the `.c` file. Then enable the 68881 options under THINK C preferences.



Customer Communication

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

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

Electronic Services

Bulletin Board Support

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

United States: (512) 794-5422 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

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.

FaxBack Support

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

(512) 418-1111 or (800) 329-7177

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
LabWindows: lw.support@natinst.com
LabVIEW: lv.support@natinst.com
HiQ: hiq.support@natinst.com
VISA: visa.support@natinst.com

Telephone and Fax Support

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

	Telephone	Fax
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	519 622 9311
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 71 11
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 48301892	02 48301915
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 (include version number) _____

Clock speed _____ MHz RAM _____ MB Display adapter _____

Mouse ___ yes ___ no Other adapters installed _____

Hard disk capacity _____ MB Brand _____

Instruments used _____

_____

National Instruments hardware product model_____ Revision _____

Configuration _____

National Instruments software product _____ Version_____

Configuration _____

The problem is: _____

_____

_____

_____

_____

List any error messages: _____

_____

_____

The following steps reproduce the problem:_____

_____

_____

_____

_____

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: LabVIEW® Code Interface Reference Manual

Edition Date: November 1995

Part Number: 320539C-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 _____

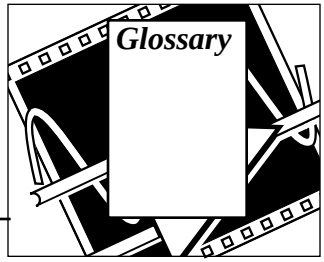
Address _____

_____

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



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

Numbers

1D — One-dimensional.

2D — Two-dimensional.

A

active window — Window that is currently set to accept user input. Usually the front window. The title bar of an active window is highlighted. You make a window active by clicking on it, or be selecting it from the **Windows** menu.

ADC — Analog-to-digital converter. An electronic device, often an integrated circuit, that converts an analog voltage to a digital number.

ANSI — American National Standards Institute.

application zone — *See* AZ.

array — Ordered, indexed set of data elements of the same type.

array shell — Front panel object that houses an array. It consists of an index display, a data object window, and an optional label. It can accept various data types.

artificial data dependency	Condition in a dataflow programming language in which the arrival of data rather than its value triggers execution of a node. *See also* data dependency.
asynchronous execution	Mode in which multiple processes share processor time, one executing while the others, for example, wait for interrupts, as while performing device I/O or waiting for a clock tick.
auto-indexing	Capability of loop structures to disassemble and assemble arrays at their borders. As an array enters a loop with auto-indexing enabled, the loop automatically disassembles it with scalars extracted from one dimensional arrays, one dimensional arrays extracted from two dimensional arrays, and so on. Loops assemble data into arrays as they exit the loop according to the reverse of the same procedure.
autoscaling	Ability of scales to adjust to the range of plotted values. On graph scales, this feature determines maximum and minimum scale values, as well.
AZ (application zone)	Memory allocation section that holds all data in a VI except execution data.

B

block diagram	Pictorial description or representation of a program or algorithm. In LabVIEW, the block diagram, which consists of executable icons called nodes and wires that carry data between the nodes, is the source code for the virtual instrument. The block diagram resides in the block diagram of the VI.
Boolean controls and indicators	Front panel objects used to manipulate and display or input and output Boolean (True or False) data. Several styles are available, such as switches, buttons and LEDs.
breakpoint	Mode that halts execution when a subVI is called. You set a breakpoint by clicking on the toolbar and then on a node.
broken VI	VI that cannot be compiled or run; signified by a run button with a broken arrow.
Bundle node	Function that creates clusters from various types of elements.

C

C string (CStr)	A series of zero or more unsigned characters, terminated by a zero, used in the C programming language.
case	One subdiagram of a Case Structure.
Case Structure	Conditional branching control structure, which executes one and only one of its subdiagrams based on its input. It is the combination of the IF THEN ELSE and CASE statements in control flow languages.
cast	To change the type descriptor of a data element without altering the memory image of the data.
chart	*See* scope chart, strip chart, and sweep chart.
CIN	*See* Code Interface Node.
CIN source code	Original, uncompiled text code. *See* object code.
Cloning	To make a copy of a control or some other LabVIEW object by <*Key*>-clicking on it and dragging the copy to its new location. In Windows, click on the object with the left mouse button while holding down the <Ctrl> key and drag the copy to its new location. On the Macintosh, <option>-click on the object and drag the copy to its new location. On the Sun, click the left mouse button while holding down the <meta> key, and drag the copy to its new location, or click on the object with the middle mouse button and drag the copy.
cluster	A set of ordered, unindexed data elements of any data type including numeric, Boolean, string, array, or cluster. The elements must be all controls or all indicators.
cluster shell	Front panel object that contains the elements of a cluster.
Code Interface Node	Special block diagram node through which you can link conventional, text-based code to a VI.
code resource	Resource that contains executable machine code. You link code resources to LabVIEW through a CIN.
coercion	The automatic conversion LabVIEW performs to change the numeric representation of a data element.
coercion dot	Glyph on a node or terminal indicating that the numeric representation of the data element changes at that point.
Color tool	Tool you use to color objects and backgrounds.

compile	Process that converts high-level code to machine-executable code. LabVIEW automatically compiles VIs before they run for the first time after creation or alteration.
concatenated Pascal string (CPStr)	A list of Pascal-type strings concatenated into a single block of memory.
connector	Part of the VI or function node that contains its input and output terminals, through which data passes to and from the node.
connector pane	Region in the upper right corner of a front panel window that displays the VI's connector. It underlies the Icon pane.
constant	*See* universal and user-defined constants.
control	Front panel object for entering data to a VI interactively or to a subVI programmatically.
control flow	Programming system in which the sequential order of instructions determines execution order. Most conventional text-based programming languages, such as C, Pascal, and BASIC, are control flow languages.
Controls palette	Menu of controls and indicators.
conversion	Changing the type of a data element.
CPStr	*See* concatenated Pascal string
current VI	VI whose front panel, block diagram, or Icon Editor is the active window.
custom PICT controls and indicators	Controls and indicators whose parts can be replaced by graphics you supply.

D

data acquisition	Process of acquiring data, typically from A/D or digital input plug-in boards.
data dependency	Condition in a dataflow programming language in which a node cannot execute until it receives data from another node. *See also* artificial data dependency.
data flow	Programming system consisting of executable nodes in which nodes execute only when they have received all required input data and produce output automatically when they have executed. LabVIEW is a dataflow system.

data logging	Generally, to acquire data and simultaneously store it in a disk file. LabVIEW file I/O functions can also log data.
data space zone	*See* DS zone.
data type descriptor	Code that identifies data types, used in data storage and representation.
dB	Decibels.
Description box	Online documentation for a LabVIEW object.
diagram window	VI window that contains the VI's block diagram code.
dimension	Size and structure attribute of an array.
DS (data space) zone	Memory allocation section that holds VI execution data.
DUT	Device under test.

E

empty array	Array that has zero elements, but has a defined data type. For example, an array that has a numeric control in its data display window but has no defined values for any element is an empty numeric array.
EOF	End-of-file. Character offset of the end of file relative to the beginning of the file (that is, the EOF is the size of the file).
executable	A stand-alone piece of code that will run, or execute.
execution highlighting	Feature that animates VI execution to illustrate the data flow in the VI.
external routine	*See* shared external routine.

F

flattened data	Data of any type that has been converted to a string, usually for writing it to a file.
For Loop	Iterative loop structure that executes its subdiagram a set number of times. Equivalent to conventional code: `For i=0 to n-1, do …`.

Formula Node	Node that executes formulas that you enter as text. Especially useful for lengthy formulas that would be cumbersome to build in block diagram form.
frame	Subdiagram of a Sequence Structure.
free label	Label on the front panel or block diagram that does not belong to any other object.
front panel	The interactive user interface of a VI. Modeled from the front panel of physical instruments, it is composed of switches, slides, meters, graphs, charts, gauges, LEDs, and other controls and indicators.
function	Built-in execution element, comparable to an operator, function, or statement in a conventional language.

G

G	LabVIEW graphical programming language.
global variable	Non-reentrant subVI with local memory that uses an uninitialized shift register to store data from one execution to the next. The memory of copies of these subVIs is shared and thus can be used to pass global data between them.
GMT	Greenwich Mean Time.
GPIB	General Purpose Interface Bus is the common name for the communications interface system defined in ANSI/IEEE Standard 488.1-1987 and ANSI/IEEE Standard 488.2-1987. Hewlett-Packard, the inventor of the bus, calls it the HP-IB.
graph control and indicator	Front panel object that displays data in a Cartesian plane.

H

handle	Pointer to a pointer to a block of memory; handles reference arrays and strings. An array of strings is a handle to a block of memory containing handles to strings.
Help window	Special window that displays the names and locations of the terminals for a function or subVI, the description of controls and indicators, the values of universal constants, and descriptions and data types of control attributes.
hierarchical menu	Menu that contains submenus or palettes.

housing	Nonmoving part of front panel controls and indicators that contains sliders and scales.
Hz	Hertz. Cycles per second.

I

icon	Graphical representation of a node on a block diagram.
Icon Editor	Interface similar to that of a paint program for creating VI icons.
icon pane	Region in the upper right-hand corner of the front panel and block diagram windows that displays the VI icon.
IEEE	Institute of Electrical and Electronic Engineers.
indicator	Front panel object that displays output.
Inf	Digital display value for a floating point representation of infinity.
inplace	Characteristic of an operation whose input and output data can use the same memory space.
instrument driver	VI that controls a programmable instrument.
I/O	Input/output.

L

label	Text object used to name or describe other objects or regions on the front panel or block diagram.
Labeling tool	Tool used to create labels and enter text into text windows.
LabVIEW	Laboratory Virtual Instrument Engineering Workbench.
LabVIEW string (LStr)	The string data type used by LabVIEW block diagrams.
legend	Object owned by a chart or graph that display the names and plot styles of plots on that chart or graph.

M

matrix	Two-dimensional array.
MB	Megabytes of memory.

mechanical-action controls and indicators	Front panel objects that look and operate like familiar mechanical or electro-mechanical devices. Examples include toggle switches, slides, meters, knobs, and LEDs
meta-click	On the Sun, to click the mouse button while pressing the <meta> key.
MPW	Macintosh Programmer's Workshop.
MSB	Most significant bit.

N

NaN	Digital display value for a floating-point representation of *not-a-number*, typically the result of an undefined operation, such as log(-1).
nodes	Execution elements of a block diagram consisting of functions, structures, and subVIs.
nondisplayable indicators	ASCII characters that cannot be displayed, such as ESC, NUL, SOH, indicators and so on.
numeric controls and indicators	Front panel objects used to manipulate and display or input and output numeric data.

O

object	Generic term for any item on the front panel or block diagram, including controls, nodes, wires, and imported pictures.
object code	Compiled version of source code. Object code is not stand-alone because you must load it into LabVIEW to run it.
Operating tool	Tool used to enter data into controls as well as operate them. Resembles a pointing finger.

P

palette	Menu that displays a palette of pictures that represent possible options.
panel window	VI window that contains the front panel, the toolbar, and the icon/connector pane.

Pascal string (PStr)	A series of unsigned characters, with the value of the first character indicating the length of the string. Used in the Pascal programming language.
plot	A graphical representation of an array of data shown either on a graph or chart.
pointer	Variable that contains an address. Commonly this address refers to a dynamically-allocated block of memory.
polymorphism	Ability of a node to automatically adjust to data of different representation, type, or structure.
pop up	To call up a special menu by clicking on an object with the right mouse button (Windows, Sun and HP-UX) or holding down the <command> key while clicking (Macintosh).
pop-up menus	Menus accessed by popping up on an object. Menu options pertain to that object specifically.
portable	Able to compile on any platform that supports LabVIEW.
Positioning tool	Tool used to move and resize objects.
private data structures	Data structures whose exact format is not described and is usually subject to change.
pull-down menus	Menus accessed from a menu bar. Menu options are usually general in nature.

R

RAM	Random Access Memory.
reentrant execution	Mode in which calls to multiple instances of a subVI can execute in parallel with distinct and separate data storage.
reference	*See* pointer.
relocatable	Able to be moved by the memory manager to a new memory location.
representation	Subtype of the numeric data type, of which there are signed and unsigned byte, word, and long integers, as well as single-, double-, and extended-precision floating-point numbers both real and complex.
ring control	Special numeric control that associates 32-bit integers, starting at 0 and increasing sequentially, with a series of text labels or graphics.

S

scalar	Number capable of being represented by a point on a scale. A single value as opposed to an array. Scalar Booleans, strings and clusters are explicitly singular instances of their respective data types.
scale	Part of mechanical-action, chart, and graph controls and indicators that contains a series of marks or points at known intervals to denote units of measure.
scope chart	Numeric indicator modeled on the operation of an oscilloscope.
sequence local	Terminal used to pass data between the frames of a Sequence Structure.
Sequence Structure	Program control structure that executes its subdiagrams in numeric order. Commonly used to force nodes that are not data dependent to execute in a desired order.
shared external routine	Subroutine that can be shared by several CIN code resources.
shift register	Optional mechanism in loop structures used to pass a variable's value from one iteration of a loop to a subsequent iteration.
sink terminal	Terminal that absorbs data. Also called a destination terminal.
slider	Moveable part of slide controls and indicators.
source code	Original, uncompiled text code.
source terminal	Terminal that emits data.
string controls and indicators	Front panel objects used to manipulate and display or input and output text.
strip chart	A numeric plotting indicator modeled after a paper strip chart recorder, which scrolls as it plots data.
structure	Program control element, such as a Sequence, Case, For Loop, or While Loop.
subdiagram	Block diagram within the border of a structure.
subVI	VI used in the block diagram of another VI; comparable to a subroutine.
sweep chart	Similar to a scope chart except a line sweeps across the screen to separate old data from new data.

T

terminal	Object or region on a node through which data passes.
tool	Special LabVIEW cursors with which you can perform specific operations.
top-level VI	VI at the top of the VI hierarchy. This term is used to distinguish the VI from its subVIs.
tunnel	Data entry or exit terminal on a structure.
type descriptor	*See* data type descriptor.

U

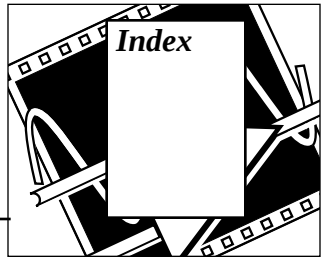
universal constant	Uneditable block diagram object that emits a particular ASCII character or standard numeric constant, for example, pi.
user-defined constant	Block diagram object that emits a value you set.
UUT	Unit under test.

V

V	Volts.
vector	One-dimensional array.
virtual instrument (VI)	LabVIEW program; so called because it models the appearance of a physical instrument.

W

While Loop	Post-iterative-test loop structure that repeats a section of code until a condition is met. Comparable to a Do loop or a Repeat-Until loop in conventional programming languages.
wire	Data path between nodes.
Wiring tool	Tool used to define data paths between source and sink terminals.



Numerics

A

B

C

D

E

F

G

H

I

L

M

N

O

P

T

U

V

W

Z