LabVIEW™

Using External Code in LabVIEW

NATIONAL INSTRUMENTS™

April 2003 Edition
Part Number 370109B-01

Important Information

Contents

About This Manual

Conventions xiii
Related Documentation xiv

Chapter 1 Introduction

Calling Code in Various Platforms 1-1
Characteristics of the Two Calling Approaches 1-2
    Details of the Call Library Function Node 1-3
    Details of a CIN 1-4
Using the Flatten To String Function 1-4

Chapter 2 Shared Libraries (DLLs)

Configuring the Call Library Function Node 2-1
    Configuring for Multiple Thread Operation 2-2
    Setting the Calling Convention 2-3
    Configuring Parameters 2-3
        Configuring Return Type 2-4
        Adding and Deleting Parameters 2-4
        Editing Parameters 2-4
        Selecting the Parameter Type 2-4
    Calling Functions That Expect Other Data Types 2-8
Building a Shared Library (DLL) 2-8
    Task 1: Build the Function Prototype in LabVIEW 2-9
    Task 2: Complete the .c File 2-11
        Required Libraries 2-12
    Task 3: Build a Library Project in an External IDE 2-13
        Microsoft Visual C++ 6.0 on 32-bit on Windows Platforms 2-13
        Gnu C or C++ Compilers on Solaris or Linux 2-16
        Metrowerks CodeWarrior on Mac OS 2-17
        Project Builder on Mac OS X 2-17
Calling External APIs 2-17
    Common Pitfalls with the Call Library Function Node 2-17
        Incorrect Function Name 2-18
        Data Types 2-19
        Constants 2-19
        Calling Conventions 2-19

Example 1: Call a Shared Library that You Built 2-20
Complete Configuration of the Call Library Function Node 2-20
Build the Front Panel 2-21
Complete the Block Diagram 2-21
Run the VI 2-22
Example 2: Call a Hardware Driver API 2-22
Configure the Call Library Function Node 2-23
Example 3: Call the Win32 API 2-24
Constants 2-26
Determining the Proper Library and Function Name 2-28
Unicode Versions and ANSI Versions of Functions 2-28
Configuring a Call to the Win32 API 2-29
Additional Windows Examples of LabVIEW Calls to DLLs 2-30
Debugging DLLs and Calls to DLLs 2-31
Troubleshooting the Call Library Function Node 2-31
Troubleshooting your DLL 2-32
Troubleshooting Checklist 2-34
Module Definition Files 2-36
Array and String Options 2-37
Arrays of Numeric Data 2-37
String Data 2-38
Array and String Tip 2-40

Chapter 3
CINs

Supported Languages 3-1
Mac OS X 3-1
Mac OS Classic 3-2
Microsoft Windows 3-2
Solaris and Linux 3-2
Resolving Multithreading Issues 3-2
Making LabVIEW Recognize a CIN as Thread Safe 3-3
Using C Code that is Thread Safe 3-3
Creating a CIN 3-4
Step 1. Set Up Input and Output Terminals for the CIN 3-4
Input-Output Terminals 3-5
Output-Only Terminals 3-6
Step 2. Wire the Inputs and Outputs to the CIN 3-6
Step 3. Create a .c File 3-7
Step 4. Compile the CIN Source Code 3-9
Mac OS X 3-9
Mac OS Classic 3-10
Microsoft Windows 3-15

Solaris 2.*x* ... 3-17
Linux ... 3-18
gcc Compiler ... 3-18
Step 5. Load the CIN Object Code ... 3-19
LabVIEW Manager Routines ... 3-19
Pointers as Parameters ... 3-20
Debugging External Code ... 3-22
DbgPrintf ... 3-22
Windows ... 3-22
UNIX ... 3-24

Chapter 4
Programming Issues for CINs

Passing Parameters ... 4-1
Parameters in the CIN .c File ... 4-1
Passing Fixed-Size Data to CINs ... 4-2
Scalar Numerics ... 4-2
Scalar Booleans ... 4-2
Refnums ... 4-2
Clusters of Scalars ... 4-2
Return Value for CIN Routines ... 4-3
Examples with Scalars ... 4-3
Passing Variably-Sized Data to CINs ... 4-7
Alignment Considerations ... 4-7
Arrays and Strings ... 4-8
Paths ... 4-8
Clusters Containing Variably-Sized Data ... 4-8
Resizing Arrays and Strings ... 4-9
Examples with Variably Sized Data ... 4-9
Manager Overview ... 4-18
Data Types ... 4-18
Scalar ... 4-19
char ... 4-21
Dynamic ... 4-21
Memory-Related ... 4-23
Constants ... 4-23
Memory Manager ... 4-23
Memory Allocation ... 4-24
Memory Zones ... 4-25
File Manager ... 4-28
Identifying Files and Directories ... 4-29
Path Specifications ... 4-29

File Descriptors 4-32
File Refnums 4-32
Support Manager 4-32

Chapter 5
Advanced Applications

CIN Routines 5-1
Data Spaces and Code Resources 5-1
One Reference to the CIN in a Single VI 5-3
Loading a VI 5-3
Unloading a VI 5-4
Loading a New Resource into the CIN 5-4
Compiling a VI 5-4
Running a VI 5-5
Saving a VI 5-5
Aborting a VI 5-5
Multiple References to the Same CIN in a Single VI 5-6
Multiple References to the Same CIN in Different VIs 5-7
Single-Threaded Operating Systems 5-7
Multithreaded Operating Systems 5-8
Code Globals and CIN Data Space Globals 5-9
Code Globals and CIN Data Space Globals Examples 5-10
Using Code Globals 5-11
Using CIN Data Space Globals 5-12

Chapter 6
Function Descriptions

Memory Manager Functions 6-1
File Manager Functions 6-2
Permissions for Files and Directories 6-2
File Manager Functions and Operations 6-3
Support Manager Functions 6-6
Mathematical Operations 6-8
Individual Function Descriptions 6-9
Abs 6-10
ASCIITime 6-11
AZCheckHandle/DSCheckHandle 6-12
AZCheckPtr/DSCheckPtr 6-13
AZCopyHandle/DSCopyHandle 6-14
AZDisposeHandle/DSDisposeHandle 6-15
AZDisposePtr/DSDisposePtr 6-16
AZGetHandleSize/DSGetHandleSize 6-17

AZHeapCheck/DSHeapCheck 6-18
AZHLock 6-19
AZHNoPurge 6-20
AZHPurge 6-21
AZHUnlock 6-22
AZMaxMem/DSMaxMem 6-23
AZMemStats/DSMemStats 6-24
AZNewHandle/DSNewHandle 6-25
AZNewHClr/DSNewHClr 6-26
AZNewPClr/DSNewPClr 6-27
AZNewPtr/DSNewPtr 6-28
AZRecoverHandle/DSRecoverHandle 6-29
AZSetHandleSize/DSSetHandleSize 6-30
AZSetHSzClr/DSSetHSzClr 6-31
BinSearch 6-32
BlockCmp 6-33
Cat4Chrs 6-34
ClearMem 6-35
CPStrBuf 6-36
CPStrCmp 6-37
CPStrIndex 6-38
CPStrInsert 6-39
CPStrLen 6-40
CPStrRemove 6-41
CPStrReplace 6-42
CPStrSize 6-43
CToPStr 6-44
DateCString 6-45
DateToSecs 6-46
FAddPath 6-47
FAppendName 6-48
FAppPath 6-49
FArrToPath 6-50
FCopy 6-51
FCreate 6-52
FCreateAlways 6-55
FDepth 6-58
FDestroyPath 6-59
FDirName 6-60
FDisposeRefNum 6-61
FEmptyPath 6-62
FExists 6-63
FFlattenPath 6-64
FFlush 6-65

FGetAccessRights 6-66
FGetDefGroup 6-67
FGetEOF 6-68
FGetInfo 6-69
FGetPathType 6-71
FGetVolInfo 6-72
FileNameCmp 6-73
FileNameIndCmp 6-74
FileNameNCmp 6-75
FIsAPath 6-76
FIsAPathOfType 6-77
FIsAPathOrNotAPath 6-78
FIsARefNum 6-79
FIsEmptyPath 6-80
FListDir 6-81
FLockOrUnlockRange 6-83
FMakePath 6-85
FMClose 6-86
FMOpen 6-87
FMove 6-90
FMRead 6-91
FMSeek 6-92
FMTell 6-93
FMWrite 6-94
FName 6-95
FNamePtr 6-96
FNewDir 6-97
FNewRefNum 6-98
FNotAPath 6-99
FPathCmp 6-100
FPathCpy 6-101
FPathToArr 6-102
FPathToAZString 6-103
FPathToDSString 6-104
FPathToPath 6-105
FRefNumToFD 6-106
FRefNumToPath 6-107
FRelPath 6-108
FRemove 6-109
FSetAccessRights 6-110
FSetEOF 6-111
FSetInfo 6-112
FSetPathType 6-114
FStrFitsPat 6-115

FStringToPath.......6-116
FTextToPath.......6-117
FUnFlattenPath.......6-118
FVolName.......6-119
GetALong.......6-120
HexChar.......6-121
Hi16.......6-122
HiByte.......6-123
HiNibble.......6-124
IsAlpha.......6-125
IsDigit.......6-126
IsLower.......6-127
IsUpper.......6-128
Lo16.......6-129
LoByte.......6-130
Long.......6-131
LoNibble.......6-132
LStrBuf.......6-133
LStrCmp.......6-134
LStrLen.......6-135
LToPStr.......6-136
Max.......6-137
MilliSecs.......6-138
Min.......6-139
MoveBlock.......6-140
NumericArrayResize.......6-141
Occur.......6-143
Offset.......6-144
Pin.......6-145
PostLVUserEvent.......6-146
PPStrCaseCmp.......6-147
PPStrCmp.......6-148
Printf.......6-149
PStrBuf.......6-152
PStrCaseCmp.......6-153
PStrCat.......6-154
PStrCmp.......6-155
PStrCpy.......6-156
PStrLen.......6-157
PStrNCpy.......6-158
PToCStr.......6-159
PToLStr.......6-160
QSort.......6-161
RandomGen.......6-162

SecsToDate 6-163
SetALong 6-164
SetCINArraySize 6-165
StrCat 6-166
StrCmp 6-167
StrCpy 6-168
StrLen 6-169
StrNCaseCmp 6-170
StrNCmp 6-171
StrNCpy 6-172
SwapBlock 6-173
TimeCString 6-174
TimeInSecs 6-175
ToLower 6-176
ToUpper 6-177
Unused 6-178
Word 6-179

Appendix A
Technical Support and Professional Services

Glossary

About This Manual

This manual describes the Call Library Function Node and the Code Interface Node (CIN). The Call Library Function Node and the CIN are the LabVIEW programming objects you use to call compiled code from text-based programming languages. This manual includes reference information about libraries of functions, memory and file manipulation routines, and diagnostic routines that you can use with calls to external code.

Conventions

The following conventions appear in this manual:

»	The » symbol leads you through nested menu items and dialog box options to a final action. The sequence **File»Page Setup»Options** directs you to pull down the **File** menu, select the **Page Setup** item, and select **Options** from the last dialog box.
💡	This icon denotes a tip, which alerts you to advisory information.
📝	This icon denotes a note, which alerts you to important information.
⚠	This icon denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.
bold	Bold text denotes items that you must select or click in the software, such as menu items and dialog box options. Bold text also denotes parameter names.
italic	Italic text denotes variables, emphasis, a cross reference, or an introduction to a key concept. This font also denotes text that is a placeholder for a word or value that you must supply.
`monospace`	Text in this font denotes text or characters that you should enter from the keyboard, sections of code, programming examples, and syntax examples. This font is also used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, functions, operations, variables, filenames and extensions, and code excerpts.
`monospace bold`	Bold text in this font denotes the messages and responses that the computer automatically prints to the screen. This font also emphasizes lines of code that are different from the other examples.

`monospace italic`	Italic text in this font denotes text that is a placeholder for a word or value that you must supply.
Platform	Text in this font denotes a specific platform and indicates that the text following it applies only to that platform.

Related Documentation

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

- *LabVIEW User Manual*
- *LabVIEW Help*
- *Using LabVIEW with TCP/IP and UDP* Application Note
- *Using Apple Events and the PPC Toolbox to Communicate with LabVIEW Applications on the Macintosh* Application Note
- *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability* Application Note

Sun users also might find the following document helpful:

- Sun Workshop CD-ROM, Sun Microsystems, Inc., U.S.A.

Linux users also might find the following document helpful:

- *The GNU C Compiler Reference Manual*, Free Software Foundation, 1989–2000.

Windows users also might find the following documents helpful:

- 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
- Microsoft Visual C++ CD-ROM, Microsoft Corporation, Redmond, WA, 1997

1 Introduction

This manual discusses the following methods used in LabVIEW to call code written in other languages:

- Using platform-specific protocols.
- Using the Call Library Function Node to call the following types of shared libraries:
  - Dynamic Link Libraries (DLL) in Windows
  - Code Fragments on Mac OS
  - Shared Libraries on UNIX
- Creating a Code Interface Node (CIN) to call code written specifically to link to VIs.

Notes To convert an instrument driver written in LabWindows™/CVI™, select **Tools»Instrumentation»Import CVI Instrument Driver** to open the **Select a CVI Function Panel file** dialog box. In the **Select a CVI Function Panel file** dialog box, you select the function panel file to convert. After you select a front panel file, the **CVI Function Panel Converter** dialog box opens. You use the **CVI Function Panel Converter** dialog box to complete the conversion of the front panel file. Refer to the *LabVIEW Help* for more information about the **CVI Function Panel Converter** dialog box.

Refer to `cvilvsb.h` in the `cintools` folder for information about creating a LabVIEW CIN in LabWindows/CVI.

Calling Code in Various Platforms

This section describes the differences between running Windows and UNIX applications from within your VIs and running Mac OS applications from within your VIs.

(Windows and UNIX) Use the System Exec VI to run a command line from your VI. The command line can include any parameters supported by the application you want to launch.

If you can access the application through TCP/IP, you might be able to pass data or commands to the application. Refer to the documentation for the application for a description of its communication capability. If you are a LabVIEW user, refer to the *Using LabVIEW with TCP/IP and UDP* Application Note for more information about techniques for using networking VIs to transfer information to other applications. You also can use many ActiveX LabVIEW tools to communicate with other applications.

(Mac OS) Use the AppleEvent VIs to send commands between applications or to launch other applications. Apple events are a Mac-specific protocol through which applications communicate with each other. If you are a LabVIEW user, refer to the *Using Apple Events and the PPC Toolbox to Communicate with LabVIEW Applications on the Macintosh* Application Note for information about different methods for using AppleEvent VIs to launch and control other applications.

Characteristics of the Two Calling Approaches

Note In most cases, a Call Library Function Node is easier to use than a CIN. Assuming the underlying code is the same, the calling speed is the same whether you use a Call Library Function Node or a CIN.

The LabVIEW Call Library Function Node and the CIN are block diagram objects that link source code written in a conventional programming language to LabVIEW. They appear on the block diagram as icons with input and output terminals. Linking external code to LabVIEW includes the following actions:

1. You compile the source code and link it to form executable code. If you already have a compiled DLL, this step is not necessary.
2. LabVIEW calls the executable code when the Call Library Function Node or CIN executes.
3. LabVIEW passes input data from the block diagram to the executable code.
4. LabVIEW returns data from the executable code to the block diagram.

The LabVIEW compiler can generate code fast enough for most programming tasks. Call CINs and shared libraries from LabVIEW to accomplish tasks a text-based language can accomplish more easily, such as time-critical tasks. Also use CINs and shared libraries for tasks you cannot perform directly from the block diagram, such as calling system routines for which no corresponding LabVIEW functions exist. CINs and shared libraries also can link existing code to LabVIEW, although you might need to modify the code so it uses the correct LabVIEW data types.

CINs and shared libraries execute synchronously, so LabVIEW cannot use the execution thread used by these objects for any other tasks. When a VI runs, LabVIEW monitors the user interface, including the menus and keyboard. In multithreaded applications, LabVIEW uses a separate thread for user interface tasks. In single-threaded applications, LabVIEW switches between user interface tasks and running VIs.

When CIN or shared library object code executes, it takes control of its execution thread. If an application has only a single thread of control, the application waits until the object code returns. In single-threaded operating systems such as Mac OS, CINs and shared libraries even prevent other applications from running.

LabVIEW cannot interrupt object code that is running, so you cannot reset a VI that is running a CIN or shared library until execution completes. If you want to write a CIN or shared library that performs a long task, be aware that LabVIEW cannot perform other tasks in the same thread while these objects execute.

Details of the Call Library Function Node

You can call most standard shared libraries with the Call Library Function Node. In Windows, these shared libraries are DLLs; on Mac OS, they are Code Fragments; and on UNIX, they are Shared Libraries. The Call Library Function Node includes a large number of data types and calling conventions. You can use the Call Library Function Node to call functions from most standard and custom-made libraries.

DLLs have the following advantages:

- You can change the DLL without changing any of the VIs that link to the DLL, provided you do not modify the function prototypes.
- Practically all modern development environments provide support for creating DLLs, while LabVIEW supports only a subset of development environments for creating CINs.

The Call Library Function Node is most appropriate when you have existing code you want to call, or if you are familiar with the process of creating standard shared libraries. Because a library uses a format standard among several development environments, you can use almost any development environment to create a library that LabVIEW can call. Refer to the documentation for your compiler to determine whether you can create standard shared libraries. Refer to Chapter 2, *Shared Libraries (DLLs)*, for more information about the Call Library Function Node.

Details of a CIN

The CIN is a general method for calling C code from LabVIEW. You can pass arbitrarily complex data structures to and from a CIN. In some cases, you might have higher performance using CINs because data structures pass to the CIN in the same format that they are stored in LabVIEW.

CINs have the following advantages:

- The CIN code is integrated into the code of the VI, so there is no extra file to maintain when the VI is distributed.
- CINs provide certain special entry points, such as `CINLoad`, `CINSave`, and so on.

In some cases, you might want a CIN to perform additional tasks at certain execution times. For example, you might want to initialize 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 ongoing task, such as accumulating results from call to call, so you can allocate, initialize, and deallocate resources at the correct time. Most CINs perform a specific action at run-time only.

To create a CIN, you must be an experienced C developer. Also, because CINs are tightly coupled with LabVIEW, restrictions exist about which compilers you can use.

After you write 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, regardless of the development system.

Using the Flatten To String Function

The Flatten To String function takes LabView data and converts it into a string. This string, when used in conjunction with the various communication functions or I/O functions, can be stored in a file or database or be sent to other computers.

The string created by the Flatten To String function is a LabVIEW string. LabVIEW strings have a 4-byte number, which is a 32-bit, signed integer data type, at the beginning of the string that specifies the length of the string. Specifying the length of the string enables a LabVIEW string to include `NULL` characters, such as the ASCII character zero (0). If a LabVIEW string is passed to external code and used as a C string, `NULL` characters embedded in the string might cause problems because C strings are interpreted as terminating at the first `NULL` character.

To pass the flattened form of LabVIEW data to C code, convert the flattened LabVIEW data from a binary string format to an alphanumeric string format. For example, suppose your string consists of the following five characters:

```
* character 35 (#)
* character 65 (A)
* character 0 (NULL)
* character 50 (2)
```

Complete the following steps to convert the preceding five characters from a binary string format to an alphanumeric string format.

1. Convert the decimal values of the five characters into hexadecimal values.

   35d = 0x23
   65d = 0x41
   0d = 0x00
   50d = 0x32
   107d = 0x6B

2. Write down the actual alphanumeric characters for the hexadecimal values and include only a single `NULL` value at the end.

   ```
   * character 50 (2)
   * character 51 (3)
   * character 52 (4)
   * character 49 (1)
   * character 48 (0)
   * character 48 (0)
   * character 51 (3)
   * character 50 (2)
   * character 54 (6)
   * character 66 (B)
   * character 0 (NULL)
   ```

Converting from a binary string format to an alphanumeric string format doubles the size of the string. However, converting to an alphanumeric format preserves the information in the string when you use the string in an environment where you have to replace LabVIEW strings with C strings.

2

Shared Libraries (DLLs)

This chapter describes how to call shared libraries from LabVIEW.

(Windows) A shared library is called a DLL. This manual uses DLL as a generic abbreviation for shared library.

(Mac OS) A shared library is called a Code Fragment.

(UNIX) A shared library is called a Shared Library function.

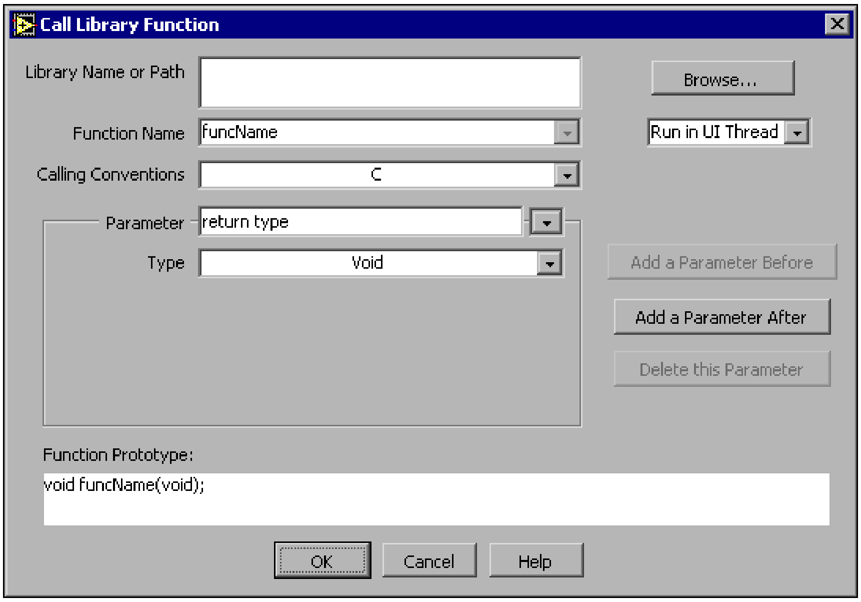
You can use any language to write DLLs as long as the DLLs can be called using one of the calling conventions LabVIEW supports, either `stdcall` or `C`. Examples and troubleshooting information appear later in the chapter to help you build and use DLLs and to successfully configure the Call Library Function Node in LabVIEW. The general methods described here for DLLs also apply to other types of shared libraries.

Refer to the `examples\dll` directory for examples of using shared libraries.

Configuring the Call Library Function Node

Use the Call Library Function Node to directly call a 32-bit Windows DLL, a Mac OS Code Fragment, or a UNIX Shared Library function.

Right-click the Call Library Function Node and select **Configure** from the shortcut menu to open the **Call Library Function** dialog box, shown in Figure 2-1. Use the **Call Library Function** dialog box to specify the library, function, parameters, return value for the object, and calling conventions in Windows. When you click **OK** in the **Call Library Function** dialog box, LabVIEW updates the Call Library Function Node according to your settings, displaying the correct number of terminals and setting the terminals to the correct data types.



Function Name: funcName

Calling Conventions: C

Parameter: return type

Type: Void

Function Prototype:

```
void funcName(void);
```

Figure 2-1. Call Library Function Dialog Box

As you configure parameters, the **Function Prototype** area displays the C prototype for the function you are building. This area is a read-only display.

Configuring for Multiple Thread Operation

In a multithreaded operating system, you can make multiple calls to a DLL or shared library simultaneously. By default, all call library objects run in the user interface thread. The control below the **Browse** button in the **Call Library Function** dialog box reflects your selection of **Run in UI Thread** or **Reentrant**.

Before you configure a Call Library Function Node to be reentrant, make sure that multiple threads can call the function simultaneously. The following characteristics are the basic characteristics of thread safe code in a shared library:

- The code is thread safe when it does not store any global data, such as global variables, files on disk, and so on.
- The code is thread safe when it does not access any hardware. In other words, the code does not contain register-level programming.
- The code is thread safe when it does not make any calls to any functions, shared libraries, or drivers that are not thread safe.

- The code is thread safe when it uses semaphores or mutexes to protect access to global resources.
- The code is thread safe when it is called by only one non-reentrant VI.

Refer to the *Execution Properties Page* topic of the *LabVIEW Help* for more information about reentrancy. Refer to the *Using LabVIEW to Create Multithreaded VIs for Maximum Performance and Reliability* Application Note for more information about multithreading in LabVIEW.

Setting the Calling Convention

Use the **Calling Conventions** pull-down menu in the **Call Library Function** dialog box to select the calling convention for the function. The default calling convention is `C`.

(Windows) You also can use the standard Windows calling convention, `stdcall`.

Refer to the documentation for the DLL you want to call for the appropriate calling conventions.

Configuring Parameters

This section discusses the return value and how to add parameters to the Call Library Function Node.

Initially, the Call Library Function Node has no parameters and has a return type of **Void**. The return type for the Call Library Function Node returns to the right terminal of the top pair of terminals. If the return type is **Void**, the top pair of terminals is unused. Each additional pair of terminals corresponds to a parameter in the **Parameter** list of the Call Library Function Node. To pass a value to the Call Library Function Node, wire to the left terminal of a terminal pair. To read the value of a parameter after the Call Library Function Node call, wire from the right terminal of a terminal pair. Figure 2-2 shows a Call Library Function Node that has a return type of **Void**, a string parameter, and a numeric parameter.

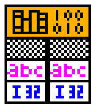


Figure 2-2. Call Library Function Node with Return Value of Void and Two Parameters

Configuring Return Type

For return type, you can set **Type** to **Void**, **Numeric**, or **String**. **Void** is only available for return type and is not available for parameters. Use **Void** for the return type if your function does not return any values.

Even if the function you call returns a value, you can use **Void** for the return type. When the function returns a value and you select **Void** as the return type, the value returned by the function is ignored.

Refer to the *Numeric* section of this chapter for information about the numeric parameter type and to the *String* section of this chapter for information about the string parameter type.

Note If the function you are calling returns a data type not listed, choose a return data type the same data size as the one returned by the function. For example, if the function returns a char data type, use an 8-bit unsigned integer. A call to a function in a DLL cannot return a pointer because there are no pointer types in LabVIEW. However, you can specify the return type as an integer that is the same size as the pointer. LabVIEW then treats the address as a simple integer, and you can pass it to future DLL calls.

Adding and Deleting Parameters

To add parameters to the Call Library Function Node, click the **Add a Parameter Before** button or the **Add a Parameter After** button. To remove a parameter, click the **Delete this Parameter** button.

Editing Parameters

Use the **Parameter** pull-down menu to select the return value or a parameter for editing. When selected, you can edit the **Parameter** name to something more descriptive, which makes it easier to switch between parameters. The **Parameter** name does not affect the call, but it is propagated to output wires. Also, you can edit all fields in the **Parameter** section for the selected parameter.

Selecting the Parameter Type

Use the **Type** pull-down menu to indicate the type of each parameter. You can select from the following parameter types:

- **Numeric**
- **Array**
- **String**
- **Waveform**

- **Digital Waveform**
- **Digital Table**
- **ActiveX**
- **Adapt to Type**

After you select an item from the **Type** pull-down menu, you see more items you can use to indicate details about the parameter and about how to pass the data to the library function. The Call Library Function Node has a number of different items for parameter types because of the variety of data types required by different libraries. Refer to the documentation for the library you call to determine which parameter types to use.

The following sections discuss the different parameter types available from the **Type** pull-down menu.

(Windows) Refer to the `examples\dll\data passing\Call Native Code.llb` for an example of using data types in shared libraries.

Numeric

For numeric data types, you must indicate the exact numeric type by using the **Data Type** pull-down menu. You can choose from the following data types:

- 8-, 16-, and 32-bit signed and unsigned integers
- 4-byte, single-precision numbers
- 8-byte, double-precision numbers

Note Extended-precision numbers and complex numbers can be passed by selecting **Adapt to Type** from the **Type** pull-down menu. However, standard libraries generally do not use extended-precision numbers and complex numbers.

Use the **Pass** pull-down menu to indicate whether you want to pass the value or a pointer to the value.

Array

Use the **Data Type** pull-down menu to indicate the data type of the array. You can choose from the same data types available for numeric parameters. Refer to the *Numeric* section of this chapter for information about numeric data types.

Specify the dimensions of the array in **Dimensions**.

Use the **Array Format** pull-down menu to make one of the following choices:

- **Array Data Pointer** passes a one-dimensional pointer to the array data.
- **Array Handle** passes a pointer to a pointer that points to a four-byte value for each dimension, followed by the data.
- **Array Handle Pointer** passes a pointer to an array handle.

Caution Do *not* attempt to resize an array with system functions, such as `realloc`. Doing so might crash your system. Instead, use one of the Code Interface Node (CIN) manager functions, such as `NumericArrayResize`. Refer to Chapter 4, *Programming Issues for CINs*, for information about CIN manager functions.

String

Use the **String Format** pull-down menu to indicate the string format. You can choose from the following string formats:

- **C String Pointer**—a string followed by a null character
- **Pascal String Pointer**—a string preceded by a length byte
- **String Handle**—a pointer to a pointer to four bytes for length information, followed by string data
- **String Handle Pointer**

Select a string format that the library function expects. Most standard libraries expect either a C string or a Pascal string. If the library function you are calling is written for LabVIEW, you might want to use the **String Handle** format.

Caution Do *not* attempt to resize a string with system functions, such as `realloc`, because your system might crash.

Waveform

When you call a shared library that includes a waveform data type, you do not have to specify a numeric value from the **Data Type** pull-down menu; the default is **8-byte double**. However, you must specify a **Dimension**. If the **Parameter** is a single waveform, specify a **Dimension** of `0`. If the **Parameter** is an array of waveforms, specify a **Dimension** of `1`. LabVIEW does not support an array of waveforms greater than 1D.

Digital Waveform

Specify a **Dimension** of `0` if the **Parameter** is a single digital waveform. Specify a **Dimension** of `1` if the **Parameter** is an array of digital waveforms. LabVIEW does not support an array of digital waveforms greater than 1D.

Digital Table

Specify a **Dimension** of `0` if the **Parameter** is a single digital table. Specify a **Dimension** of `1` if the **Parameter** is an array of digital tables. LabVIEW does not support an array of digital tables greater than 1D.

Note Waveforms, digital waveforms, and digital tables can be passed through shared libraries but accessing the data inside the shared libraries is not supported at this time.

ActiveX

Select one of the following items from the **Data Type** pull-down menu:

- **ActiveX Variant Pointer** passes a pointer to ActiveX data.
- **IDispatch* Pointer** passes a pointer to the IDispatch interface of an ActiveX Automation server.
- **IUnknown Pointer** passes a pointer to the IUnknown interface of an ActiveX Automation server.

Adapt to Type

Use **Adapt to Type** to pass arbitrary LabVIEW data types to DLLs in the same way they are passed to a CIN. The arbitrary LabVIEW data types are passed to DLLs in the following ways:

- Scalars are passed by reference. A pointer to the scalar is passed to the library.
- Arrays and strings are passed according to the **Data Format** setting. You can choose from the following **Data Format** settings:
  - **Handles by Value** passes the handle to the library. The handle is not NULL.
  - **Pointers to Handles** passes a pointer to the handle to the library. If the handle is NULL, treat the handle as an empty string or array. To set a value when the handle is NULL, you must allocate a new handle.
- Clusters are passed by reference.

- Scalar elements in arrays or clusters are in line. For example, a cluster containing a numeric is passed as a pointer to a structure containing a numeric.
- Clusters within arrays are in line.
- Strings and arrays within clusters are referenced by a handle.

Note When one or more of the parameters of the function you want to call in a DLL are of types that do not exist in LabVIEW, ensure that each parameter is passed to the function in a way that allows the DLL to correctly interpret the data. Create a skeleton `.c` file from the current configuration of the Call Library Function Node. By viewing the `.c` file, you can determine whether LabVIEW will pass the data in a manner compatible with the DLL function. You then can make any necessary adjustments. Refer to the *Task 2: Complete the .c File* section of this chapter for information about creating a skeleton `.c` file.

Calling Functions That Expect Other Data Types

You might encounter a function that expects a data type LabVIEW does not use. For example, you cannot use the Call Library Function Node to pass an arbitrary cluster or array of nonnumeric data. If you need to call a function that expects other data types, use one of the following methods:

- Depending on the data type, you might be able to pass the data by creating a string or array of bytes that contains a binary image of the data you want to send. You can create binary data by typecasting data elements to strings and concatenating them.
- Write a library function that accepts data types that LabVIEW does use. Use parameters the library function expects to build the data structures. Call the library function.
- Write a CIN that can accept arbitrary data structures. Refer to Chapter 3, *CINs*, for more information about writing CINs.

Building a Shared Library (DLL)

Building external code libraries to call from LabVIEW consists of the following basic tasks:

- *Task 1: Build the Function Prototype in LabVIEW*
- *Task 2: Complete the .c File*
- *Task 3: Build a Library Project in an External IDE*

This section uses a simple shared library example to describe the three basic tasks for building external code libraries to call from LabVIEW.

In the *Example 1: Call a Shared Library that You Built* section, you call the shared library that you build here.

Task 1: Build the Function Prototype in LabVIEW

To build a function prototype for your shared library, you must build a prototype in LabVIEW and then fill in all the details of your code. When you allow LabVIEW to generate this C source code, you help ensure that the basic syntax of the code in your shared library is valid. The prototype source file you create is a `.c` file and contains C declarations for the parameters you want to pass.

Complete the following steps to build your prototype source file, `myshared.c`.

1. Create a new VI named Array Average.
2. Place a Call Library Function Node on the block diagram.
3. Right-click the Call Library Function Node icon and select **Configure** from the shortcut menu to open the **Call Library Function** dialog box.
4. Leave the **Library Name or Path** field empty.

Note Use the **Library Name or Path** field to specify the shared library the Call Library Function Node calls. For this example, you provide the file path in the *Example 1: Call a Shared Library that You Built* section.

5. Enter the following general specifications:
   a. Type `avg_num` in the **Function Name** field.
   b. Select **C** from the **Calling Conventions** pull-down menu.
6. Define the return value using the following specifications:
   a. Change the default name in the **Parameter** field from `return type` to the more descriptive name `error`.
   b. Select **Numeric** from the **Type** pull-down menu.
   c. Select **Signed 32-bit Integer** from the **Data Type** pull-down menu.
7. Define the `a` parameter using the following specifications:
   a. Click the **Add Parameter After** button.
   b. Replace the default name `arg1` in the **Parameter** field with the precise name, `a`.

   c. Select **Array** from the **Type** pull-down menu.
   d. Select **4-byte Single** from the **Data Type** pull-down menu.
   e. Select **Array Data Pointer** from the **Array Format** pull-down menu.

Note The *Array and String Options* section describes the available settings for arrays and strings in the Call Library Function Node icon.

8. Define the `size` parameter using the following specifications:
   a. Click the **Add Parameter After** button.
   b. Replace the default name `arg2` in the **Parameter** field with the precise name, `size`.
   c. Select **Numeric** from the **Type** pull-down menu.
   d. Select **Signed 32-bit Integer** from the **Data Type** pull-down menu.
   e. Select **Value** from the **Pass** pull-down menu.
9. Define the `avg` parameter using the following specifications:
   a. Click the **Add Parameter After** button.
   b. Replace the default name `arg3` in the **Parameter** field with the precise name, `avg`.
   c. Select **Numeric** from the **Type** pull-down menu.
   d. Select **4-byte Single** from the **Data Type** pull-down menu.
   e. Select **Pointer to Value** from the **Pass** pull-down menu.
10. Check that the Function Prototype field displays the return value and three parameters in the correct order, as follows:

```
long avg_num(float *a, long size, float *avg);
```

Note The syntax you see in the **Function Prototype** field is technically correct. However, the `.c` file that LabVIEW generates in the next section is more precise because the first parameter appears as **`float a[]`**.

11. Click the **OK** button to save your settings and close the dialog box.
12. Notice how the Call Library Function Node icon updates to reflect your settings.
13. Right-click the Call Library Function Node icon and select **Create .c file** from the shortcut menu.
14. Save the file as `myshared.c`.

Note In this example, you use a `.c` source file. When you work with C++ libraries, change the extension of the source file to `.cpp`.

Preventing C++ Name Decoration

When you build shared libraries for C++, you must prevent the C++ compiler from decorating the function names in the final object code. To do this, wrap the function declaration in an `extern "C"` clause, as shown in the following prototype:

```
extern "C" {
long MyDLLFunction(long nInput, unsigned long nOutput,
                 void *arg1);
}

long MyDLLFunction(long nInput, unsigned long nOutput,
                 void *arg1)
    {

    /* Insert Code Here */

    }
```

Note If you disable C++ decoration of a function, the compiler cannot create polymorphic versions of the function.

Task 2: Complete the .c File

The Call Library Function Node generates the following source code skeleton in `myshared.c`:

```
/* Call Library Source File */

#include "extcode.h"

long avg_num(float a[], long size, float *avg);

long avg_num(float a[], long size, float *avg)

        {

        /* Insert Code Here */

        }
```

Replace the `/* Insert Code Here */` spacer with the following function code, making sure to place the code within the pair of curly braces:

```
    int i;
    float sum = 0;
```

```
if(a != NULL)
    {
    for(i=0; i < size; i++)
        sum = sum + a[i];
    }
else
    return (1);
*avg = sum / size;
return (0);
```

Required Libraries

This simple example requires no header files. When you build more complex shared libraries, you must include header files for all related libraries. For example, a Windows shared library project might need to include `windows.h`. In another instance, a project might need to include `extcode.h`, the header file for the set of LabVIEW manager functions that perform simple and complex operations, ranging from low-level byte manipulation to routines for sorting data and managing memory.

When you want to use the LabVIEW manager functions inside your shared library, you must include the following LabVIEW library files in your compiled project:

- `labview.lib` for Visual C++
- `labview.sym.lib` for Symantec
- `labview.export.stub` for Metrowerks CodeWarrior

The preceding LabVIEW library files appear in the `cintools` directory of your LabVIEW installation. Specifically, you need the LabVIEW manager functions if you intend to do any of the following tasks:

- Allocate, free, or resize arrays, strings, or other data structures that are passed into or out of your library from LabVIEW.
- Work with LabVIEW Path data types.
- Work with file refnums inside your library.
- Use any of the Support Manager functions.

Refer to Chapter 6, *Function Descriptions*, for more information about the manager functions.

Task 3: Build a Library Project in an External IDE

The process of building a library project is specific to each integrated development environment (IDE) and each operating system. This section describes the following compiler/platform combinations that you can use to build shared libraries to use in LabVIEW:

- Microsoft Visual C++ on Windows
- Gnu C/C++ on UNIX
- Metrowerks CodeWarrior on Mac Classic
- Apple's Project Builder for Mac OS X

Microsoft Visual C++ 6.0 on 32-bit on Windows Platforms

This section discusses how to set up a project that compiles `myshared.c` and generates `myshared.dll`.

Adding the DLL Export Keyword

You must explicitly export each function from your DLL to make it available to LabVIEW. For this example, you should use the `_declspec (dllexport)` keyword to export the `avg_num` function. `_declspec (dllexport)` is a Microsoft-specific extension to the C or C++ language. By declaring the dllexport keyword, you eliminate the need for a module definition file. Refer to the *Module Definition Files* section of this chapter for information about module definition files.

Open `myshared.c` and insert the `_declspec(dllexport)` keyword in front of the code for `avg_num`. This function also has a declaration statement, and you must place the keyword in front of the declaration, too.

The following excerpt shows the two places in `myshared.c` that require the `_declspec(dllexport)` keyword:

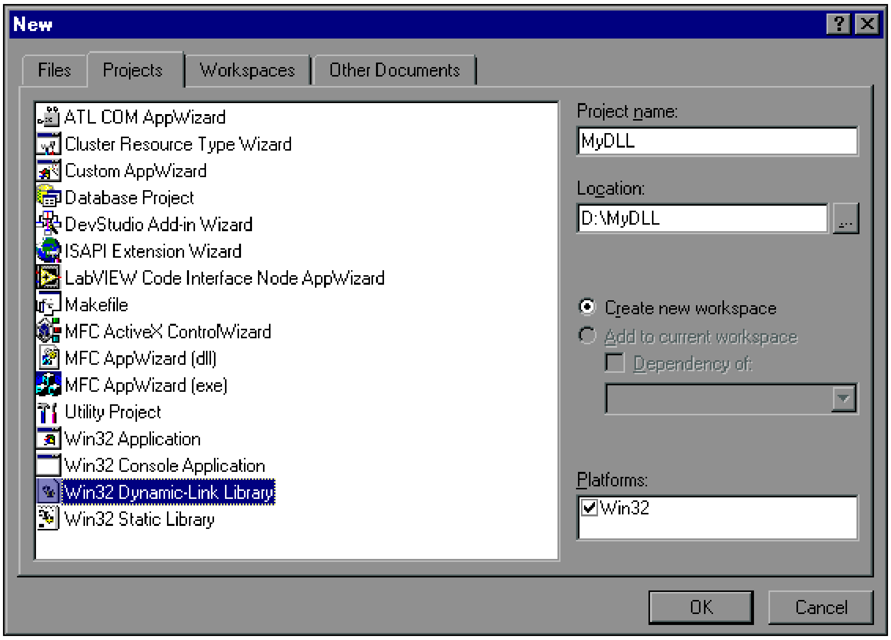
```
_declspec(dllexport) long avg_num(float a[],
                               long size, float *avg);
_declspec(dllexport) long avg_num(float a,
                               long size, float *avg)
```

Setting Up the Project

You set up your project in the Microsoft Visual C++ integrated development environment. Complete the following steps to set up a project for `myshared.c`.

1. Select **File»New** and select **Win32 Dynamic Link Library** (DLL) in the listbox on the **Projects** tab, as shown in Figure 2-3.
2. Click the **OK** button.

Note You do not use Microsoft Foundation Classes (MFC) in this example. However, if you want to use these object classes in a project, you can select **MFC AppWizard (dll)** at this point, instead of selecting **Win32 Dynamic Link Library**. Then, copy the code from the `myshared.c` source file and place it into the skeleton source code file that the MFC AppWizard generates.



Project name: MyDLL

Location: D:\MyDLL

[● Create new workspace]

Platforms: ☑ Win32

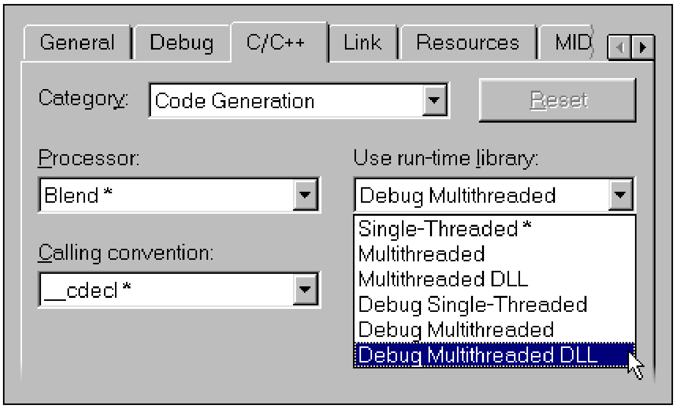
Figure 2-3. Creating a Project in Visual C++

3. Select **An empty DLL project** when prompted to choose the type of DLL that you want to create.
4. Click the **Finish** button to finish creating your project and return to the Visual C++ workspace.

5. Select **Add to Project»Files** from the **Project** menu and add the `myshared.c` source file.

Note When you want to use the LabVIEW manager functions in a Windows DLL, you also must add `labview.lib` to your project. The `cintools` directory of your LabVIEW installation contains this `.lib` file.

6. Select **Project»Settings** and click the **C++** tab of the **Project Settings** dialog box and make the following settings:
   a. Select **Code Generation** from the **Category** pull-down menu.
   b. Set the **Struct member alignment** control to **1 Byte** for this example and for all configurations.
   c. Select **Debug Multithreaded DLL** from the **Use run-time library** pull-down menu to apply the Win32 Debug configuration, as shown in Figure 2-4.



Category: Code Generation

Processor: Blend *

Calling convention: __cdecl *

Use run-time library: Debug Multithreaded

Single-Threaded *
Multithreaded
Multithreaded DLL
Debug Single-Threaded
Debug Multithreaded
Debug Multithreaded DLL

Figure 2-4. Setting the **Use run-time library** Control, Microsoft Visual C++

   You have the option to choose the Win32 Release configuration, instead. In that case, you select **Multithreaded DLL** in the **Use run-time library** control.

7. Select **Build»Build myshared.dll** to cause Visual C/C++ to build a DLL and place it in either the Debug or Release output directory, depending on which configuration option you selected in step 6c.

In the *Example 1: Call a Shared Library that You Built* section, you call this DLL from LabVIEW.

Gnu C or C++ Compilers on Solaris or Linux

Use the following command to compile the `myshared.c` source file that you completed in the *Task 2: Complete the .c File* section:

```
gcc -fPIC -shared -o <output name> <source file>
```

The `-fPIC` option instructs GCC to produce position-independent code, which is suitable for shared libraries. The `-shared` option specifies that the output should be a shared library file.

Note Some versions of the Gnu linker do not produce suitable output for shared libraries. The `-fno-gnu-linker` instructs GCC to use the system linker rather than the Gnu linker. The output name is normally a file with a `.so` extension on Solaris and Linux.

Reducing Symbol Scope

By default, all symbols (functions and global variables) defined in your code are available. It is sometimes desirable for your library to distinguish between those symbols that should be accessed by external objects and those that are for internal use only. Use a mapfile to make these distinctions. The mapfile is a text document that the linker takes as input and uses to determine, among other things, which symbols should be exported.

Use the following basic syntax for a mapfile, where <*library file*> is the name of the output file:

```
<library file> {
global:
[Symbol for global scope 1];
[Symbol for global scope 2];
...
local:
[Symbols for local scope 1]; or "*"
...
};
```

Under the global and local sections, list all of the symbols that you want to be available globally or locally, respectively. Each section is optional, but remember that all symbols are global by default. In the local section, you can choose to use the `"*"` wildcard rather than listing individual symbols. This wildcard means, “any symbol not already defined as global” and allows you to easily make symbol definitions in terms of symbols to be exported rather than symbols to be reduced in scope.

After you create the mapfile, save it and instruct the linker to use it by appending `-M <mapfile>` to the gcc command-line argument list.

Metrowerks CodeWarrior on Mac OS

Create a shared library using the process that the Metrowerks documentation describes. To use this shared library with LabVIEW, you must set **struct alignment** to **68k** in the PPC Processor settings panel. Be sure to export the function(s) that you want to call from LabVIEW.

Project Builder on Mac OS X

Create a framework using the process described in the Project Builder documentation. You must include the **-malign-natural** setting in the **Other C Compiler Flags** section of the **GCC Compiler Settings**. If you want to call functions in LabVIEW from your framework, you need to link against `liblvexports.a`, located in `cintools/Mach-O`.

Calling External APIs

You might need to access external APIs from within LabVIEW code. Most often, you access external APIs to obtain functionality that the operating system provides. Normally, you can use the LabVIEW Call Library Function Node to accomplish this goal. You must provide the following information to the Call Library Function Node excess external APIs from within Labview Code:

- Function name as it appears in the library
- Function prototype
- Library or module in which the function resides
- Calling conventions of the function
- Thread-safe status of the function

Common Pitfalls with the Call Library Function Node

The function reference documentation for any API should provide most of the information that the Call Library Function Node requires. However, you should keep in mind the common errors listed in this section.

Incorrect Function Name

Your library call can fail when the name of the function as it appears in the library is different than is expected. Usually this error occurs due to function name redefinition, or to function name decoration, as in the following examples:

- **Redefinition**—This pitfall appears when an API manufacturer uses a define mechanism, such as `#define` directive in ANSI C, to define an abstracted function name to one of many functions present in the library, based on some external condition such as language or debug mode. In such cases, you can look in the header (`.h`) file for the API to determine whether a `#define` directive redefined the name of a function you want to use.
- **Function Name Decoration**—This pitfall appears when certain functions have their names decorated when they are linked. A typical C compiler tracks name decoration, and when it looks for a function in a shared library, it recognizes the decorated name. However, because LabVIEW is not a C compiler, it does not recognize decorated names. If you suspect that function name decoration is causing difficulty, inspect the shared library's exported functions.

Note If the function name that appears in the function prototype section has characters such as @ appended to it, the function was decorated when the DLL was built. This is most common with C++ compilers.

In LabVIEW, the **Function Name** control in the **Call Library Function** dialog box is a pull-down list where you can access a list of all functions within the library you have selected. In addition, most operating systems have a utility you can use to view a library's exports, for example, QuickView on the Windows operating system and the `nm` command on most UNIX systems.

Note **(Windows and Mac)** If the **Function Name** list contains entries but the function you want to call does not appear in the list, the most likely reason is that the function has not been exported. Refer to the documentation for your compiler for information about how to mark functions for export. **(UNIX)** The **Function Name** list always appears as an empty list. You must enter the name of the function you want to call.

Data Types

Your library call can fail when you do not use the correct data types. LabVIEW only supports basic numeric data types and C strings. Also, you can select **Adapt to Type** from the **Type** pull-down menu of the **Call Library Function** dialog box and direct LabVIEW to pass its own internal data types for a given parameter. You might encounter the following specific problems:

- **Non-Standard Data Type Definitions**—Frequently, other APIs use non-standard definitions for data types. For example, instead of using `char`, `short`, and `long`, the Windows API uses `BYTE`, `WORD`, and `DWORD`. If an API that you are using makes use of such data types, you need to find the equivalent basic C data type so that you can properly configure the Call Library Function Node. The *Example 3: Call the Win32 API* section presents an example of this process.
- **Structure and Class Data Types**—Some APIs have structure and, in the case of C++, class data types. LabVIEW cannot use these data types. If you need to use a function that has a structure or class as an argument, you should write a CIN or shared library wrapper function that takes as inputs the data types that LabVIEW supports and that appropriately packages them before LabVIEW calls the desired function.

(Windows) Refer to the `examples\dll\data passing\Call Native Code.llb` for an example of using data types in shared libraries.

Constants

Your library call can fail when your external code uses identifiers in place of constants. Many APIs define identifiers for constants to make the code easier to read. LabVIEW must receive the actual value of the constant rather than the identifier that a particular API uses. Constants are usually numeric, but they might also be strings or other values. To identify all constants, inspect the header file for the API to find the definitions. The definition might either be in `#define` statements or in enumerations, which use the `enum` keyword. The *Constants* section presents an example of this identification process.

Calling Conventions

Your library call can fail when certain operating systems use calling conventions other than the C calling convention and the Standard (`__stdcall`) calling convention. The calling convention defines how data is passed to a function and how clean up occurs after the function call is complete. The documentation for the API should say which calling convention(s) you must use. The Standard (`__stdcall`) calling convention is also known as the WINAPI convention, or the Pascal convention.

Use of calling conventions other than the C or Standard calling conventions frequently causes the failure of library calls in LabVIEW because those other calling conventions use an incompatible method for maintaining the stack.

Example 1: Call a Shared Library that You Built

This section describes the tasks necessary to complete the Array Average VI you started building in the *Building a Shared Library (DLL)* section so the VI can call the `avg_num` function in `myshared.dll`. **(UNIX)** The shared library file has a `.so` or `.sl` extension. The following tasks must be completed before the Array Average VI can call the `avg_num` function in `myshared.dll`:

- Complete configuration of the Call Library Function Node.
- Build the front panel.
- Complete the block diagram.

Complete Configuration of the Call Library Function Node

Complete the following steps to complete the configuration of the Call Library Function Node.

1. Open the Array Average VI block diagram.
2. Right-click the Call Library Function Node and select **Configure** from the shortcut menu to open the **Call Library Function** dialog box.
3. Click the **Browse** button to open the **Select a library** dialog box.
4. Navigate to the location of your `myshared.dll` file.
5. Select `myshared.dll` and click the **Open** button. The file path to `myshared.dll` appears in the **Library Name or Path** field. The **Library Name or Path** field is the shared library that the Call Library Function Node calls.

Note To make the reference to your shared library platform independent, use `.*` for the file extension.

Build the Front Panel

Complete the following steps to create the front panel of the Array Average VI.

1. Place an **Array** control on the front panel and label it `Array`.
2. Place a **Numeric Control** in the array shell and resize the array to contain four elements.
3. Right-click the `Array` control and select **Reprensentation»Single Precision** from the shortcut menu.
4. Place a Numeric Indicator on the front panel and label it `Avg Value` to display the result of your averaging calculation.
5. Right-click the `Avg Value` indicator and select **Reprensentation»Single Precision** from the shortcut menu.
6. Place a Numeric Indicator on the front panel and label it `Error` to display any errors that your VI generates.
7. Right-click the `Error` indicator and select **Reprensentation»Long** from the shortcut menu.

Figure 2-5 shows the Array Average VI front panel.

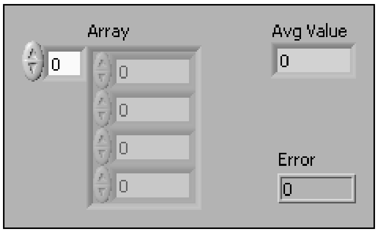


Figure 2-5. Array Average VI Front Panel

Complete the Block Diagram

Complete the following steps to complete the block diagram of the Array Average VI.

1. Wire `Array` to the `a` input of the Call Library Function Node.
2. Place an Array Size function on the block diagram.
3. Wire `Array` to the input of the Array Size function.
4. Wire the Array Size function output to the `size` input of the Call Library Function Node.

5. Right-click the `avg` input of the Call Library Function Node and select **Create»Constant** from the shortcut menu. Set the constant value to zero.
6. Wire the `avg` output of the Call Library Function Node to `Avg Value`.
7. Wire the `error` output of the Call Library Function Node to `Error`.

Figure 2-6 shows the completed Array Average block diagram.

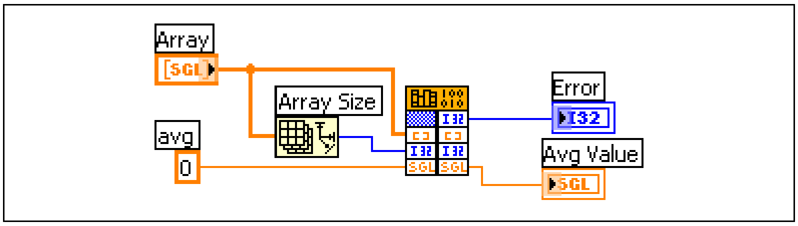


Figure 2-6. Array Average VI Block Diagram

Run the VI

On the front panel, enter values in **Array** and run the VI to calculate the average of those values. Save your work and close the VI.

If your DLL returns incorrect results or crashes, verify the data types and wiring to see if you wired the wrong type of information. If you require further help, several sections in this chapter present troubleshooting tips and pitfalls to avoid.

Example 2: Call a Hardware Driver API

You might want to access an API associated with hardware you have purchased. In this example, you call a hypothetical interface card for a databus called X-bus.

Note You do not need to use the Call Library Function Node to gain access to the APIs of National Instruments hardware. All National Instruments products come with LabVIEW interfaces.

Configure the Call Library Function Node

The X-bus interface card comes with a software driver for your operating system. The X-bus documentation provides the following standard information:

- A listing of all functions you can use to access the hardware
- Description of the shared library file `xbus.dll` that contains these functions
- Instructions on including a header file `xbus.h`

Note Although LabVIEW does not permit you to include such header files, you can open header files and extract information about function prototypes and constants.

- A statement about the Standard (`__stdcall`) calling convention that the X-bus library uses

One of the functions you want to use with this hypothetical hardware is `XBusRead16`, which reads a 16-bit integer from a certain address. The documentation describes `XBusRead16` as follows:

```
long XBusRead16(unsigned long offset, short* data);
```

Puts 16 bits from the register at “offset” into the memory location pointed to by “data.” Returns 1 if successful, or 0 if it fails.

Given the preceding information from the X-bus documentation, complete the following steps to configure the LabVIEW Call Library Function Node.

1. Create a new VI named Read Data.
2. Place a Call Library Function Node on the block diagram.
3. Right-click the Call Library Function Node object and select **Configure** from the shortcut menu.
4. Make the following settings in the **Call Library Function** dialog box:
    a. Enter `XbusRead16`, in the **Function Name** control.
    b. Select **stdcall (WINAPI)** from the **Calling Conventions** pull-down menu.
    c. Select **Numeric** from the **Type** pull-down menu for return type.
    d. Select **Signed 32-bit Integer** from the **Data Type** pull-down menu for return type.
    e. Add a parameter and name it **offset**.

f. Select **Numeric** from the **Type** pull-down menu.

g. Select **Unsigned 32-bit Integer** from the **Data Type** pull-down menu.

h. Add a parameter and name it **data**.

i. Select **Numeric** from the **Type** pull-down menu.

j. Select **Signed 16-bit Integer** from the **Data Type** pull-down menu

k. Select **Point to Value** from the **Pass** pull-down menu.

5. Inspect the function prototype that appears in the **Function Prototype** field. If the prototype you see does not match the definition of the function in the API you are calling, you must change your settings in the **Call Library Function** dialog box.

Place a **Numeric Control**, Numeric Indicator, and Round LED indicator on the front panel. Label the control and indicators and complete the block diagram as shown in Figure 2-7.

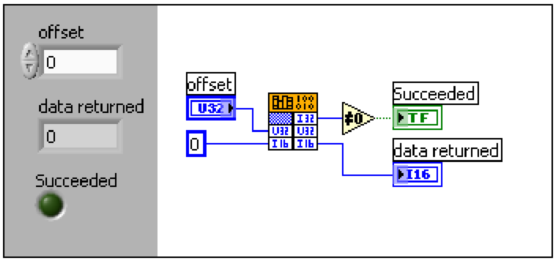


Figure 2-7. Read Data VI Front Panel and Block Diagram

Example 3: Call the Win32 API

You might want to access the 32-bit Windows platform API (Win32 API). In Win32 environments, various DLLs permit your application to interact with the operating system and with the graphical user interface. Because the API offers thousands of functions, programmers must rely on the documentation for the Microsoft Software Development Kit (SDK). Microsoft Visual Studio products give you access to the SDK documentation. You also can access this information at the Microsoft Developer Network (MSDN) Web site on the Internet.

Note Instead of using the Windows DLL as described in this example, you could easily create this message box in LabVIEW.

In this example, you call the Windows `MessageBox` function, a function which illustrates several of the typical complexities of the Win32 API. `MessageBox` is a simple SDK function that presents a small dialog box with a message, and has the following prototype:

```
int MessageBox( HWND hWnd,  // handle to owner window
             LPCTSTR lpText,  // text in message box
             LPCTSTR lpCaption,  // message box title
             UINT uType  // message box style );
```

Notice the non-standard data types, such as `HWND` and `LPCTSTR`. The Win32 API uses hundreds of data types in the SDK, and very few of them are standard C data types. However, many of the non-standard data types are merely aliases for standard C data types. The API uses the aliases to identify the context of a particular data type. Table 2-1 lists the data types in the preceding prototype and the corresponding standard C data types:

Table 2-1. Mapping Win32 Data Types to Standard C Data Types

WIN32 SDK Data Type	Basic C Data Type
`HWND`	`int **`
`LPCTSTR`	`const char *`
`UINT`	`unsigned int`

To properly call the `MessageBox` function in LabVIEW, you need to identify the equivalent LabVIEW data types, which you can usually infer from the C data types. Mapping `LPCTSTR` and `UINT` to LabVIEW is straightforward: `LPCTSTR` is a C String and `UINT` is a `U32`.

Mapping `HWND` is more complex. Table 2-1 lists `HWND` as a double pointer to an integer. However, inspection of the function shows that `MessageBox` uses `HWND` merely as a reference number that identifies the owner of the window. Because of this fact, you do not need to know the integer value for which the `HWND` is a handle. Instead, you need to know the value of the `HWND` variable itself. Because it is a double pointer, and hence a pointer, you can be treat it as a 32-bit unsigned integer, or in LabVIEW terms, a `U32`. Handles such as `HWND` are common in the Win32 SDK. In LabVIEW, you are almost always interested in the handle itself and not the data to which it points. Therefore, you can usually treat handles as `U32`. Handle names always begin with the letter H in the Win32 API.

If the SDK documentation does not make clear what C data type corresponds to a Win32 type, search `windef.h` for the appropriate `#define` or `typedef` statement.

Table 2-2 lists the Win32 SDK data types from Table 2-1 and their corresponding LabVIEW data types.

Table 2-2. Mapping Win32 Data Types to LabVIEW Data Types

WIN32 SDK Data Type	LabVIEW Data Type
`HWND`	`uInt32`
`LPCTSTR`	`CStr (C string pointer)`
`UINT`	`uInt32`

(Windows) Refer to the `examples\dll\data passing\Call Native Code.llb` for a list of more Win32 API data types.

Constants

This section presents methods for finding the numerical values of constants in the Win32 API, using `MessageBox` constants as examples. Table 2-3 lists selected constants for `MessageBox`.

Table 2-3. Selected Constants for MessageBox

Constant	Description
`MB_ABORTRETRYIGNORE`	An Abort, Retry, Ignore message box.
`MB_CANCELTRYCONTINUE`	A Cancel, Try Again, Continue message box in Windows 2000. An alternative to `MB_ABORTRETRYIGNORE`.
`MB_HELP`	A **Help** button to add to a message box for Windows 2000/XP/Me/2000, Windows NT 4.0 and later. The system sends a `WM_HELP` message to the owner whenever the user clicks the **Help** button or presses <F1>.
`MB_OK`	A message box with an **OK** button. This is the default message box.

In Visual Studio, programmers do not use the actual values of constants. In LabVIEW, however, you need to pass the actual numeric value of the constant to the function. You find these values in the header files that come with the SDK. The SDK online documentation normally lists the relevant header file at the bottom of the help topic for a given function. For `MessageBox`, the SDK online documentation has the following statement:

```
Header: Declared in winuser.h
```

The header file named in the preceding statement usually declares the constants. Searching through that header file, you should be able to find a `#define` statement or an enumeration that assigns the constant text a value. `winuser.h` defines values for some of the `MessageBox` constants as follows:

```
#define MB_OK 0x00000000L
#define MB_ABORTRETRYIGNORE 0x00000002L
#define MB_ICONWARNING MB_ICONEXCLAMATION
```

Thus, `MB_OK` has the decimal value `0`. `MB_ABORTRETRYIGNORE` has the decimal value `2`. `MB_ICONWARNING` is defined as `MB_ICONEXCLAMATION`. Elsewhere in `winuser.h` you find the following statement defining `MB_ICONEXCLAMATION`:

```
#define MB_ICONEXCLAMATION 0x00000030L
```

A hexadecimal value of `30` translates to a decimal value of `48`.

Tips Keep in mind that constants in the SDK often are used in bitfields. A bitfield is usually a single integer in which each bit controls a certain property. The `uType` parameter in `MessageBox` is an example of a bitfield. Often, you can combine multiple constants in order to set multiple properties through one parameter. In order to combine these constants, you use a bit-wise OR operation (|). For example, to set the `MessageBox` to have a warning icon and the buttons **Abort**, **Retry**, and **Ignore**, you pass the following value of `uType` to `MessageBox`:

```
MB_ABORTRETRYIGNORE | MB_ICONEXCLAMATION = 0x32
```

In LabVIEW, you combine multiple constants by wiring integer types to the OR operator, as shown in Figure 2-8.

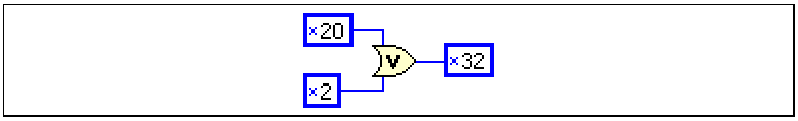


Figure 2-8. Combining Function Constants in LabVIEW

Determining the Proper Library and Function Name

Before you can configure the call to the Win32 API, you must identify the DLL that contains `MessageBox` and the specific name of `MessageBox` within the DLL. Refer to the description of `MessageBox` in the documentation that comes with your SDK or search for “MessageBox” on the Microsoft Web site. A *Requirements* section follows the function description for `MessageBox` and contains the following information:

“Requirements:

Windows NT: Requires version 3.1 or later.

Windows: Requires Windows 98 or later.

Windows CE: Requires version 1.0 or later.

Header: Declared in winuser.h.

Import Library: Use user32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows and Windows NT.”

The Import Library line names the static library `user32.lib` that you need to link to in order to build a program in the C language. Every static library in the SDK has a dynamic counterpart that has the same file name but a `.dll` extension instead of a `.lib` extension. The DLL contains the actual implementation of the desired function. So, in this case, you know that `user32.dll` contains `MessageBox`.

Unicode Versions and ANSI Versions of Functions

`MessageBox` uses two string arguments. The SDK implements two versions of functions that use string arguments, a Unicode version and an ANSI version. One of the items in the *Requirements* section of the `MessageBox` documentation says, “Unicode: Implemented as Unicode and ANSI version on Windows and Windows NT.” You can distinguish the two versions in the DLL because each has a `W` (Unicode) or an `A` (ANSI) appended to the end of the function name. `winuser.h` contains the following code:

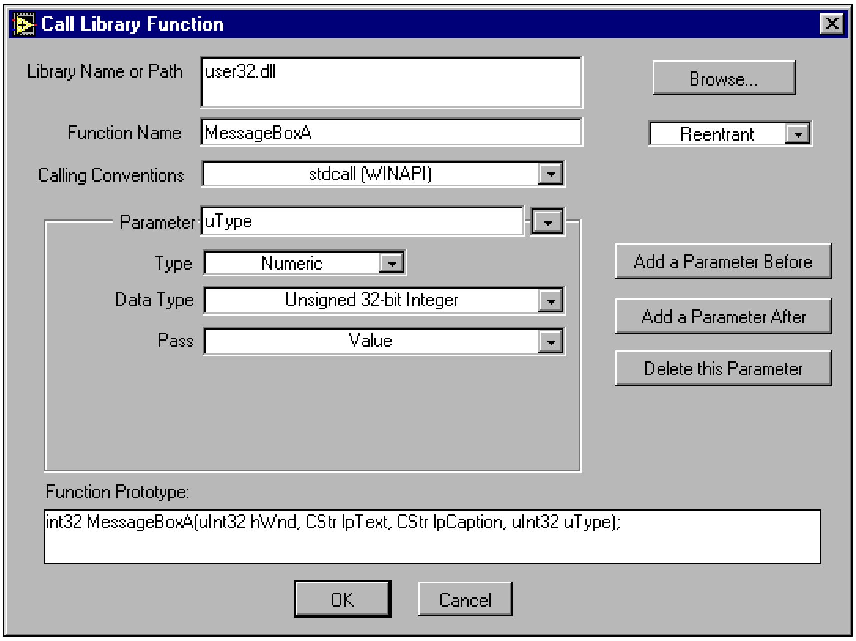
```
#ifdef UNICODE
#define MessageBox  MessageBoxW
#else
#define MessageBox  MessageBoxA
#endif // !UNICODE
```

The preceding code defines `MessageBox` to be either `MessageBoxA` or `MessageBoxW`, depending on whether the application is a Unicode application. In effect, a `MessageBox` function does not exist in `user32.dll`. Instead, `user32.dll` contains a function `MessageBoxA` and a function `MessageBoxW`. In most cases in LabVIEW, you use the ANSI version of the function because the LabVIEW strings are based on ANSI, not Unicode. For this example, you use the `MessageBoxA` function.

Configuring a Call to the Win32 API

Now that you are familiar with many aspects of the Win32 API, you can configure a LabVIEW Call Library Function Node to call the `MessageBox` function. Remember that you must use the Standard (`__stdcall`) calling convention in calls to any function in the Windows SDK.

Figure 2-9 shows a correctly configured instance of the Call Library Function Node. Make your **Call Library Function** dialog box match the settings in the graphic. Refer to the *Task 1: Build the Function Prototype in LabVIEW* section of this chapter for a separate example that teaches you how to configure controls for the Call Library Function Node.


Call Library Function

Library Name or Path: user32.dll

Function Name: MessageBoxA

Reentrant

Calling Conventions: stdcall (WINAPI)

Parameter: uType

Type: Numeric

Data Type: Unsigned 32-bit Integer

Pass: Value

Function Prototype:

int32 MessageBoxA(uInt32 hWnd, CStr lpText, CStr lpCaption, uInt32 uType);

Figure 2-9. Configuring the Call Library Function Node to Call the Win32 API

Figure 2-10 shows the block for a VI designed to call the Win32 API. Configure your block diagram to match Figure 2-10.

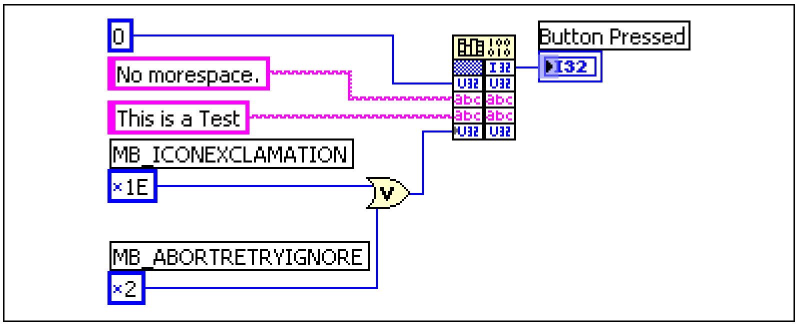
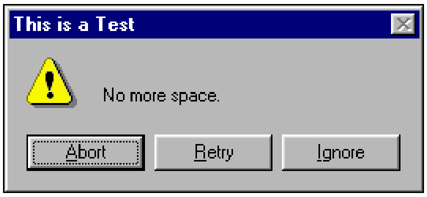


Figure 2-10. Block Diagram for a Call to the Win32 API

The VI generates the message box shown in Figure 2-11.



No more space.

Figure 2-11. Running a LabVIEW Call to the Win32 API

Additional Windows Examples of LabVIEW Calls to DLLs

(Windows) The following examples can help you learn more about calling DLLs from LabVIEW.

- The Call DLL VI located in `examples\dll\data passing\Call Native Code.llb` allows you to browse examples of C and C++ external code data types and how they interface with LabVIEW.
- The Play Sound VI located in the `examples\dll\sound\playsnd.llb\Play Sound.vi` directory lets you play Windows `.WAV` sound files on your computer from LabVIEW, if you have a sound card with Windows sound drivers installed on your system.
- If you do not have a sound, card you can generate a sound in your PC speaker by calling the `MessageBeep` function in `User32.DLL`. The function prototype is as follows:

```
VOID MessageBeep(UINT uType);
```

- The Hostname VI located in the `examples\dll\hostname\hostname.vi` directory returns the host name of your computer, demonstrating how to use LabVIEW string handles.
- You can programmatically position your cursor anywhere on your monitor using the `SetCursorPos` function in `User32.DLL`. The function prototype is as follows:

  ```
  BOOL SetCursorPos(INT x, INT y);
  ```

  `x` and `y` are the coordinates you want, referenced from the upper left corner of the screen. The return value is TRUE if the function was successful and FALSE if it was unsuccessful. Remember that the value returned is type BOOL, which is defined in the Win32 API as a 32-bit signed integer with values 0 = FALSE and 1 = TRUE.

Debugging DLLs and Calls to DLLs

When you debug your LabVIEW calls to DLLs, you must be prepared to trace problems in the DLL you are calling and in your implementation of the Call Library Function Node in LabVIEW.

Troubleshooting the Call Library Function Node

When your LabVIEW calls to DLLs generate errors, check for the following problems in your use of the Call Library Function Node.

- Make sure the path to the DLL file is correct.
- Make sure the version of your DLL matches the version of LabVIEW, for example, a 16-bit DLL instead of a 32-bit LabVIEW DLL.
- If LabVIEW gives you the error message **`function not found in library`**, double-check your spelling of the name of the function you want to call. Remember that function names are case sensitive. Also, be sure that your compiler has not decorated the function. Refer to the *Preventing C++ Name Decoration* section of this chapter for information about name decoration.
- If you receive an error message that a secondary DLL cannot be found, yet you properly specified the path to the primary DLL in the Call Library Function Node, the primary DLL needs additional functions from one or more other DLLs. You need to find the other DLLs and place them in the same directory as the DLL that needs them or in a directory that is in the search path. Refer to KnowledgeBase document 1F39A18U, *LabVIEW Error Message: A Secondary DLL Cannot Be Found*, at `ni.com` for more information about finding missing DLLs.

- If your VI crashes, make sure that you are passing exactly the parameters that the function in the DLL expects. For example, make sure that you are passing an `int16` and not an `int32` when the function expects `int16`. Check for errors in the code of the DLL, such as dereferencing a null pointer. Also, confirm that you are using the correct calling convention, `__stdcall` or `C`.
- Make sure all the parameters are defined to be passed by the correct method, such as value or pointer.
- If you receive a **`memory.cpp error`** message, the cause is almost always an error in the code of the DLL, such as writing past the end of the memory allocated for an array. Notice that these kinds of crashes might or might not occur at the time the DLL call actually executes on the block diagram.

Refer to the *Troubleshooting your DLL* and the *Troubleshooting Checklist* sections of this chapter for more information about troubleshooting calls to DLLs.

Troubleshooting your DLL

Check for the following problems in your DLL when LabVIEW calls to DLLs generate errors:

- Remember that you need to declare the function with the `_declspec (dllexport)` keyword in the header file and the source code, or define it in the exports section of the module definition file.
- When you use the `_declspec (dllexport)` keyword and you are also using the `__stdcall` calling convention, you must declare the DLL function name in the EXPORTS section of the module definition (`.def`) file. In the absence of a `.def` file, `__stdcall` might truncate function names in an unpredictable pattern, so the actual function name is unavailable to applications that call the DLL.
- When a function has not been properly exported, you must recompile the DLL. Before recompiling, you must close all applications and VIs that might make use of the DLL. Otherwise, the recompile will fail because the DLL is still in memory. Most compilers warn you when the DLL is in use by an application.
- After you confirm the name of the function, and after you confirm proper export of the function, find out whether you have used the C or C++ compiler on the code. If you have used the C++ compiler, the names of the functions in the DLL are altered by a process called name mangling. The easiest way to correct name mangling is to enclose the

declarations of the functions you want to export in your header file with the `extern "C"` statement, as shown in the following example code:

```
extern "C"
{
/* your function prototypes here */
}
```

- Try to debug your DLL by using the source level debugger provided with your compiler. Using the debugger of your compiler, you can set breakpoints, step through your code, watch the values of the variables, and so on. Debugging using conventional tools can be extremely beneficial. Refer to the appropriate manual for your compiler for more information about debugging.
- Calling the DLL from another C program is also another way to debug your DLL. By calling the DLL from another C program, you have a means of testing your DLL independent of LabVIEW, thus helping you to identify any problems, sooner.
- When calling a LabVIEW DLL that passes a 2D array, you must first declare the handler variable and initialize the variable to `NULL`, as shown in the following C code:

```
main ()
{
    /* Labview data handler variable for the array */
    TD1Hd1 myArray = NULL;
        .
        .
        .
    /* Call to the Labview DLL function */
    DLLFunctionalCall (&myAray) ;
        .
        .
        .
}
```

If you do not initialize the handler variable to `NULL`, the code produces a **`General Protection Fault`** when you call the DLL.

Refer to the *Troubleshooting the Call Library Function Node* and the *Troubleshooting Checklist* sections of this chapter for more information about troubleshooting calls to DLLs.

Troubleshooting Checklist

Complete the following checklist to eliminate many potential problems from LabVIEW VIs that call DLLs.

- ❑ The Call Library Function Node uses the proper calling convention (C or `__stdcall`).
- ❑ The version of your DLL matches the version of LabVIEW.
- ❑ Call Library Function Node has the correct path to the DLL.
- ❑ DLLs providing secondary DLLs additional functions are in the same directory as the DLL needing the additional functions or are in a directory that is in the search path.
- ❑ The Call Library Function Node has the correct spelling, syntax, and case sensitivity for the function name that you are calling. Otherwise, the error message **`Function not found in library`** appears.
- ❑ Data is wired to the input terminals of all the parameters of the Call Library Function Node that you are passing to a DLL function. Also, check that the Library Function Node is properly configured for all input parameters.
- ❑ Return types and data types of arguments for functions in the Call Library Function Node exactly match the data types your function uses. Erroneous data type assignments can cause crashes.
- ❑ The Call Library Function Node passes arguments to the function in the correct order.
- ❑ Resizing of arrays and concatenation of strings can take place only under the following conditions:
  - Only when the Call Library Function Node passes a LabVIEW Array Handle or LabVIEW String Handle
  - Only when you add `labview.lib` to a Visual C++ project, `labview.export.stub` to a CodeWarrior project, and `labview.sym.lib` to a Symantec project

⚠ **Caution** Never resize arrays or concatenate strings using the arguments passed directly to a function. Remember, the parameters you pass are LabVIEW data. Changing array or string sizes might result in a crash by overwriting other data stored in LabVIEW memory.

❑ The Call Library Function Node passes strings of the correct type to a function: C string pointers, Pascal string pointers, or the LabVIEW string handles. The Windows API requires the C-style string pointer.

❑ All parameters are defined to be passed by the correct method, such as value or pointer.

❑ Pascal strings do not exceed 255 characters in length.

❑ Remember that C strings are `NULL` terminated. If your DLL function returns numeric data in a binary string format, for example, through GPIB or the serial port, it might return `NULL` values as part of the data string.

❑ For arrays or strings of data, always pass a buffer or array that is large enough to hold any results that the function places in the buffer. However, if you are passing them as LabVIEW handles, use CIN functions to resize them under Visual C++, CodeWarrior, or Symantec compilers.

❑ When you are using `__stdcall`, list DLL functions in the `EXPORTS` section of the module definition file.

❑ DLL functions that other applications call appear in the module definition file `EXPORTS` section, or you include the `_declspec (dllexport)` keyword in the function declaration.

❑ When you use a C++ compiler, export functions with the `extern "C"{}` statement in your header file in order to prevent name mangling.

❑ For a DLL that you have written, never recompile the DLL while the DLL is loaded into memory by another application, for example, by your VI. Before recompiling a DLL, make sure that all applications making use of the DLL are unloaded from memory. This ensures that the DLL itself is not loaded into memory during a recompile. The DLL might fail to rebuild correctly if you forget this point and your compiler does not warn you.

❑ Test the DLL with another program to ensure that the function and the DLL behave correctly. Testing it with the debugger of your compiler or a simple C program in which you can call a function in a DLL can help you identify whether possible difficulties are inherent to the DLL or are related to LabVIEW.

Module Definition Files

In the *Building a Shared Library (DLL)* section, you configured LabVIEW to use the C calling convention in the `.c` source file you built with the Call Library Function Node. In contrast, you use the `__stdcall` calling convention when you call the Win32 API. When you build a shared library (DLL) with `__stdcall`, you normally use a module definition (`.def`) file to export the functions in your DLL. In the absence of a `.def` file, `__stdcall` might truncate function names in an unpredictable pattern, so the actual function name would be unavailable to applications that call the DLL.

You can associate a `.def` file with a DLL. The `.def` file contains the statements for defining a DLL, such as the name of the DLL and the functions that it exports, as shown in the following example code:

```
LIBRARY myshared
    EXPORTS
        avg_num
```

The preceding code example demonstrates the following key requirements for `.def` files:

- The only mandatory entries in the `.def` files are the `LIBRARY` statement and the `EXPORT` statement.
- The `LIBRARY` statement must be the first statement in the file.
- The name you specify in the `LIBRARY` statement identifies the library in the import library of the DLL.
- The names you specify in the `EXPORTS` statement identify the functions that the DLL exports.

Note Instead of a `.def` file, many Windows programmers use the **LINK** option in Project Settings of the Visual C++ compiler to obtain equivalent command-line options for most module definition statements.

Array and String Options

This section reviews important concepts regarding array and string data in the Call Library Function Node.

(Windows) Refer to the `examples\dll\data passing\Call Native Code.llb` for an example of using arrays and strings in shared libraries.

Arrays of Numeric Data

Arrays of numeric data can be comprised of any type of integers or floating point numbers with single (4-byte) or double (8-byte) precision. When you pass an array of data to a DLL function, you can pass the data as an Array Data Pointer, as a LabVIEW Array Handle, or as a LabVIEW Array Handle Pointer.

Array Data Pointers have the following characteristics whether you pass the Array Data Pointers in the Windows API or in another API.

- You can set the number of dimensions in the array, but you must not include information about the size of the array dimension(s). Instead, you must pass the size of the array dimension(s) information to your DLL in a separate variable.
- Never resize an array or perform operations that might change the length of the array data passed from LabVIEW. Resizing might cause a crash because the pointer sent is not an allocated block but points into the middle of an allocated block.
- To return an array of data, you should allocate an array of sufficient size in LabVIEW, pass the array to your function, and have this array act as the buffer. If the data takes less space, you can return the correct size as a separate parameter and then, on the calling diagram, use array subset to extract the valid data.

Remember that the Windows API does not use LabVIEW array handles, so with functions that are part of the Windows API you can use only Array Data Pointers.

If you pass the array data as a LabVIEW Array Handle, you can use LabVIEW CIN functions to resize the array. In order to call LabVIEW CIN functions, your compile must include the correct LabVIEW library file, which is located within the LabVIEW `cintools` directory. Table 2-4 lists different compilers and the correct LabVIEW library file to use with the compiler.

Table 2-4. Compilers and the LabVIEW Library File Used with Each Compiler

Compiler	LabVIEW Library File
CodeWarrior	`labview.export.stub`
Symantec	`labview.sym.lib`
Visual C++	`labview.lib`

String Data

The types of your string pointers must match the types of string pointers that your function uses, or errors occur. The Call Library Function Node offers the following choices:

- **C String Pointer** is a pointer to the string, followed by a NULL character. Most Win32 API functions use this C-style string pointer.
- **Pascal String Pointer** is a pointer to the string, preceded by a length byte.
- **LabVIEW String Handle** is a pointer to a pointer to the string, preceded by four bytes of length information.
- **LabVIEW String Handle Pointer** is a pointer to a handle for a string, preceded by four bytes of length information.

You can think of a string as an array of characters. Assembling the characters in order forms a string. LabVIEW stores a string in a special format in which the first four bytes of the array of characters form a 32-bit signed integer that stores how many characters appear in the string. Thus, a string with *n* characters requires *n* + 4 bytes to store in memory. For example, in Figure 2-12, the string `text` contains four characters.

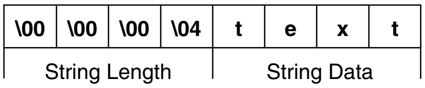


Figure 2-12. LabVIEW String Format

When LabVIEW stores the string, the first four bytes contain the value `4` as a 32-bit signed number, and each of the following four bytes contains a character of the string. The advantage of this type of string storage is that NULL characters are allowed in the string. Strings are virtually unlimited in length, up to 2^{31} characters. This method of string storage is illustrated in Figure 2-12. If you pass a LabVIEW String Handle from the Call Library

Function Node to the DLL, then you can use the LabVIEW CIN functions, such as `DSSetHandleSize`, to resize the LabVIEW String Handle.

Remember, you must add the correct LabVIEW library file to your project. Refer to Table 2-4 for a list of LabVIEW library files and the compilers with which they are used.

The Pascal string format is nearly identical to the LabVIEW string format, but instead of storing the length of the string as a 32-bit signed integer, the string length is stored as an 8-bit unsigned integer. Storing the string length as an 8-bit unsigned integer limits the length of a Pascal-style string to 255 characters. A Pascal string that is *n* characters long will require *n* + 1 bytes of memory to store. Figure 2-13 illustrates a Pascal string.

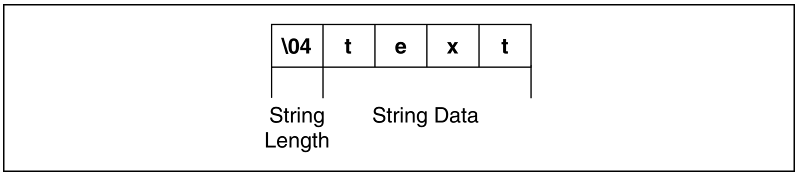


Figure 2-13. Pascal String Format

C strings are probably the type of strings you will deal with most commonly. The similarities between the C-style string and normal numeric arrays in C becomes much more clear when you see that C strings are declared as `char *`, where `char` is typically an 8-bit unsigned integer. Unlike LabVIEW and Pascal strings, C strings do not contain any information that directly gives the length of the string. Instead, C strings use a special character called the NULL character to indicate the end of the string. NULL is defined to have a value of zero in the ASCII character set. Notice that NULL is the number zero and not the character "`0`." Thus, in C, a string containing *n* characters requires *n* + 1 bytes of memory to store: *n* bytes for the characters in the string and one additional byte for the NULL termination character. Figure 2-14 illustrates how a C-style string is stored in memory.

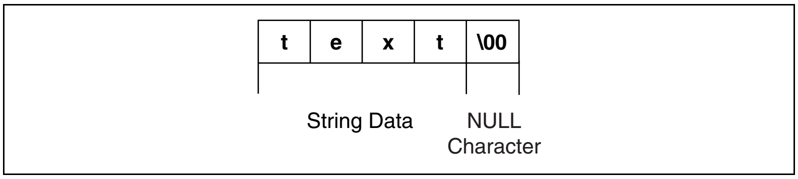


Figure 2-14. C String Format

The advantage of C-style strings is that they are limited in size only by available memory. However, if you are acquiring data from an instrument that returns numeric data as a binary string, as is common with serial or GPIB instruments, values of zero in the string are possible. If you treat the string as a C-style string, your program incorrectly assumes that the end of the string has been reached, when in fact your instrument is returning a numeric value of zero. For binary data that might contain NULL values, consider using an array of 8-bit unsigned integers.

Observe the following guidelines when passing string data to a DLL.

- Never resize a string, concatenate a string, or perform operations that might increase the length of string data passed from LabVIEW if you are using the C or Pascal string pointers.
- If you must return data as a string, allocate a string of the appropriate length in LabVIEW and pass this string into the DLL to act as a buffer.
- If you pass a LabVIEW String Handle from the Call Library Function Node to the DLL, you can use the LabVIEW CIN functions, such as `DSSetHandleSize`, to resize the LabVIEW string handle.

  In order to call LabVIEW CIN functions, your compile must include the correct LabVIEW library file, which is located within the LabVIEW `cintools` directory. Table 2-4 lists different compilers and the correct LabVIEW library file to use with the compiler.

Array and String Tip

When you are not passing LabVIEW handles and your DLL function must create an array, resize an array, or resize a string of data, you should break the DLL function into the following two DLL functions:

- Determine the number of elements that the array requires or the length of the string to be returned. Have the first DLL function return the desired size to LabVIEW.
- In LabVIEW, initialize an array or string with default values and pass this array to the second DLL function in your DLL, which actually places the data into the array. If you are working with string-based instrument control, it might be easier to pass an array of 8-bit integers than C strings because of the possibility of having NULL values in the string.

  When you are passing a LabVIEW Array Handle or LabVIEW String Handle from the Call Library Function Node to your DLL, you can use the LabVIEW CIN functions to resize or create an array or string. Refer to the *Required Libraries* section of this chapter for more information about CIN functions.

3

CINs

This chapter discusses the LabVIEW Code Interface Node (CIN), which is a block diagram node that links C/C++ source code to LabVIEW.

Notes It is technically possible to write CINs in a language other than C or C++ if the CIN entry points, such as `CINRun`, `CINLoad`, and so on, are declared as `extern "C"`. However, National Instruments recommends using a shared library (DLL) rather than a CIN if you want to use a language other than C or C++. Refer to Chapter 2, *Shared Libraries (DLLs)*, for information about DLLs.

LabVIEW does not support the creation of external subroutines. If you want to share code among multiple CINs, use DLLs. Refer to Chapter 2, *Shared Libraries (DLLs)*, for information about shared libraries.

Supported Languages

The interface for CINs 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 appropriate for a specific platform. The code must be relocatable because LabVIEW loads external code into the same memory space as the main application.

Mac OS X

You can create CINs with the Project Builder development environment, available free from Apple Computer, Inc.

Although it is possible to create a CIN using Metrowerks Codewarrior, National Instruments currently does not provide any support for this compiler.

Refer to the *Mac OS X* subsection of the *Step 4. Compile the CIN Source Code* section of this chapter for information about creating a CIN with Project Builder.

Mac OS Classic

You can create CINs with compilers from the two major C compiler vendors.

- Metrowerks CodeWarrior from Metrowerks Corporation of Austin, TX
- Macintosh Programmer’s Workshop (MPW) from Apple Computer, Inc. of Cupertino, CA

Refer to the *Mac OS Classic* subsection of the *Step 4. Compile the CIN Source Code* section of this chapter for information about creating a CIN using these compilers.

Microsoft Windows

LabVIEW for Windows supports CINs created with the following compilers:

- Microsoft Visual C++
- Symantec C

  Refer to the *Microsoft Windows* subsection of the *Step 4. Compile the CIN Source Code* section of this chapter for information about creating a CIN using these compilers.

Solaris and Linux

The default compiler for Solaris is gcc. If gcc is not installed, cc becomes the default compiler for Solaris. The only supported compiler for Linux is gcc.

Refer to the *Solaris 2.x* and *Linux* subsections of the *Step 4. Compile the CIN Source Code* section of this chapter for information about creating a CIN on Solaris and Linux.

Resolving Multithreading Issues

You must resolve the following issues to make multithreaded CINs:

- Make LabVIEW recognize your CIN as being multithreaded.
- Use C code that is completely multithread safe.

Making LabVIEW Recognize a CIN as Thread Safe

The CIN node on the block diagram is orange if you have not set the node to be thread safe. A thread safe node is yellow. Complete the following steps to make LabVIEW recognize a CIN node as thread safe.

1. Add the `CINProperties` function to your CIN code in the prototypes section of your `.c` source file, as shown in the following example code:

```
CIN MgErr CINProperties(int32 prop, void *data);
```

2. Add the following function statement to the functions section of your `.c` source file:

```
CIN MgErr CINProperties(int32 prop, void *data)
     {
     switch (prop) {
          case kCINIsReentrant:
               *(Bool32 *)data = TRUE;
                    return noErr;
               }
     return mgNotSupported;
     }
```

Using C Code that is Thread Safe

The `CINProperties` function only labels your CIN as being safe to run from multiple threads. Whether the CIN is actually thread safe depends entirely upon what C code has been written. For information about what makes C code safe or unsafe to be run from multiple threads simultaneously, please consult C programming documentation. The following characteristics are the basic characteristics of thread safe code in a CIN.

- The code is thread safe when it does not store any unprotected global data, such as global variables, files on disk, and so on.
- The code is thread safe when it does not access any hardware. In other words, the code does not contain register-level programming.
- The code is thread safe when it does not make any calls to any functions, shared libraries, or drivers that are not thread safe.
- The code is thread safe when it uses semaphores or mutexes to protect access to global resources.
- The CIN call is thread safe when only one non-reentrant VI calls the CIN.

- The CIN call is thread safe when the code does not access any global resources through CIN housekeeping routines, such as `CINInit`, `CINAbort`, `CINDispose`, and others.

Creating a CIN

Complete the following general steps to create a CIN.

1. Describe in LabVIEW the data you want to pass to the CIN.
2. Write and compile the code for the CIN using one of the supported programming languages.
3. Run a utility that puts the compiled code into a format LabVIEW can use.
4. Instruct LabVIEW to load the CIN.

If you run the VI after completing the preceding steps and the block diagram needs to execute the CIN, LabVIEW calls the CIN object code and passes any data wired to the CIN. If you save the VI after loading the code, LabVIEW saves the CIN object code along with the VI, so 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\CINs` directory contains all of the examples used in this manual. The names of the directories in the `examples\CINs` directory correspond to the CIN name in the examples.

Complete the following specific steps to create a CIN.

1. Set up input and output terminals for the CIN.
2. Wire the inputs and outputs to the CIN.
3. Create a `.c` file.
4. Compile the CIN source code.
5. Load the CIN object code.

The following sections discuss each of the preceding steps.

Step 1. Set Up Input and Output Terminals for the CIN

Place a CIN on a block diagram.

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. To add additional terminals, you can resize the node, or right-click the node and select **Add Parameter** from the shortcut menu.

Figure 3-1 shows how to resize the CIN to add parameters.

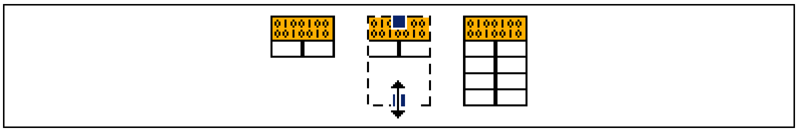


Figure 3-1. Resizing a CIN

Each pair of terminals corresponds to a parameter 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. The right terminal is the output terminal. For example, the CIN in Figure 3-2 has a single terminal pair with a 32-bit signed integer control wired to the input terminal and a 32-bit signed integer indicator wired to the output terminal.

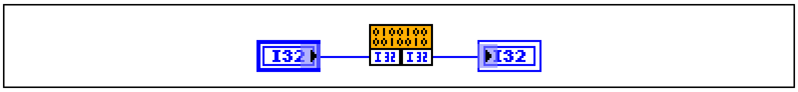


Figure 3-2. CIN with a Control and Indicator Wired to the Terminal Pair

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 signed integer input. When the CIN completes, LabVIEW then passes the value referenced by the pointer to the 32-bit signed integer indicator. When you wire controls and indicators to the input terminal and the output terminal of a terminal pair, LabVIEW assumes the CIN can modify the data passed. If another node on the block diagram needs the input value, LabVIEW might have to copy the input data before passing it to the CIN.

The CIN in Figure 3-3 has a 32-bit signed integer control wired to the input terminal but no indicator wired to the output terminal.

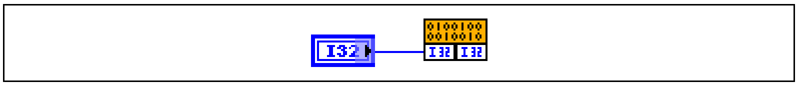


Figure 3-3. CIN with Only a Control Wired to the Terminal Pair

If you do not wire an indicator to the output terminal of a terminal pair, LabVIEW assumes the CIN does 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 might 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 resizing the CIN, right-clicking the terminal pair, and selecting **Output Only** from the shortcut menu. If a terminal pair is output-only, the input terminal is gray, as shown Figure 3-4.

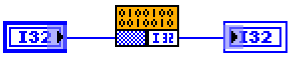


Figure 3-4. CIN with an Output-Only Terminal Pair

For output-only terminals, LabVIEW creates storage space for a return value and passes the value by reference to the CIN, the same way 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. To solve this problem, wire a control to the input terminal of the terminal pair as shown in Figure 3-4. In this case, the output terminal takes on the same data type as the input terminal. LabVIEW uses the input data 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, right-click the terminal you want to remove and select **Remove Parameter** from the shortcut 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.

Step 2. Wire the Inputs and Outputs to the CIN

Connect wires to all the terminal pairs on the CIN to specify the data you want to pass to the CIN and the data 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. You can use any LabVIEW data types as CIN parameters, so you can pass arbitrarily complex hierarchical data structures, such as arrays containing clusters that can in turn contain other arrays or clusters to a CIN. Refer to the *Passing Parameters* section of Chapter 4, *Programming Issues for CINs*, for information about how LabVIEW passes parameters of specific data types to CINs.

Step 3. Create a .c File

Right-click the CIN and select **Create .c File** from the shortcut menu to create 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, a call to a CIN takes a 32-bit signed integer as an input and returns a 32-bit signed integer as an output, as shown in Figure 3-5.

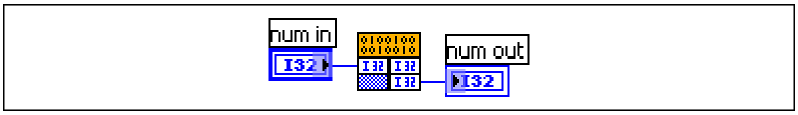


Figure 3-5. CIN with 32-bit Signed Integer Input and 32-Bit Signed Integer Output

The following code excerpt is the initial `.c` file for the CIN in Figure 3-5:

```
/* CIN source file */

#include "extcode.h"

MgErr CINRun(int32 *numIn, int32 *numOut);

MgErr CINRun(int32 *numIn, int32 *numOut) {

    /* Insert code here */
    return noErr;
    }
```

You can write eight routines for the CIN in Figure 3-5. The `CINRun` routine is required and the others are optional. If an optional routine is not present, LabVIEW uses a default routine when building the CIN.

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

Always use `#include "extcode.h"` at the beginning of your source code. If your code needs to make system calls, also use `#include "hosttype.h"` immediately after `#include "extcode.h"` and then include your system header files. `hosttype.h` includes only a subset of the `.h` files for a given operating system. If the `.h` file you need is not included in `hosttype.h`, you can include it in the `.c` file for your CIN 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`, `CINDispose`, and `CINProperties`, are housekeeping routines. The housekeeping routines are called at specific times so you can take care of specialized tasks with your CIN. For example, LabVIEW calls `CINLoad` when it first loads a VI. If you need to accomplish a special task when your VI loads, put the code for that task in the `CINLoad` routine. The following example code shows how to put the code for a task in the `CINLoad` routine:

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

In general, you only need to write the `CINRun` routine. Use the other routines when 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 example code shows how to fill out the `CINRun` routine in the `.c` file for the CIN in Figure 3-5 to multiply a number by two:

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

Refer to the *Passing Parameters* section of Chapter 4, *Programming Issues for CINs*, for information and examples about how LabVIEW passes data to a CIN.

Step 4. Compile the CIN Source Code

You must compile the source code for the CIN as a LabVIEW subroutine (`.lsb`) file. After you compile your C/C++ code in one of the compilers that LabVIEW supports, you use a LabVIEW utility that puts the object code into the `.lsb` format.

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. The specific utility you use depends on the platform and compiler you use. Refer to the following sections for more information about compiling on your platform.

Note The LabVIEW Base Development system can use existing `.lsb` files but cannot create new `.lsb` files. You can create `.lsb` files in the LabVIEW Full and Professional Development Systems.

Mac OS X

CINs compiled for LabVIEW on Mac OS Classic are not compatible with LabVIEW on Mac OS X. You must compile CINs into the Mach-O binary format and use the natural alignment of data in your compiler settings to rebuild the CINs you created in Mac OS Classic.

LabVIEW includes a template to help you build CINs using the Project Builder development environment from Apple Computer, Inc. It is possible to build a CIN with Metrowerks CodeWarrior, but LabVIEW does not provide a template or instructions.

Project Builder

To set up a CIN project for Mac OS X, you must first install the CIN template into Project Builder.

Install the CIN template into Project Builder by dragging the `LabVIEW Templates` folder from the `LabVIEW/cintools/Project Builder Files` directory into the `Project Templates` folder in the `/Developer/ProjectBuilder Extras` directory.

Complete the following steps to create a new CIN project.

1. Open Project Builder and select **File»New Project**.
2. Select **LabVIEW Templates»CIN** and specify a name and location for the project.
3. Add source code by selecting **Project»Add Files**.
4. Build a `.lsb file`. LabVIEW places the `.lsb` file next to the project file.

The LabVIEW CIN template specifies the location of the `cintools` directory in the LabVIEW directory. Project Builder assumes that you installed LabVIEW in the `Applications` directory. Complete the following steps if LabVIEW is not installed in the `Applications` directory.

1. Select **Target Settings»Expert View** and change the `LABVIEW_PATH` variable to the path to the LabVIEW directory.
2. Select **Target Settings»Search Path** and change the **Libraries Search Path** so it specifies the `cintools/Mach-O` folder in the LabVIEW directory.
3. On the **Files** tab, press **Command-I** and select the `liblabviewcin.a` file in the **External Frameworks and Libraries** section.
4. Click the **Set Path** button and locate the `liblabviewcin.a` file in the `cintools/Mach-O` folder in the LabVIEW directory.
5. On the **Files** tab, press **Command-I** and select the `liblvexports.a` file in the **External Frameworks and Libraries** section.
6. Click the **Set Path** button and locate the `liblvexports.a` file in the `cintools/Mach-O` folder in the LabVIEW directory.

The `ReadMe (Project Builder).rtf` file located in the `cintools/Project Builder Files` directory also includes information about how to install the CIN template into Project Builder and how to create a new CIN project.

Mac OS Classic

LabVIEW for Mac OS Classic uses shared libraries as a resource for customized code. To prepare the code for LabVIEW, use the separate utilities `lvsbutil.app` for Metrowerks CodeWarrior and `lvsbutil.tool` for the Macintosh Programmer's Workshop. These utilities are included with LabVIEW in the `cintools` folder.

LabVIEW header files are compatible with Metrowerks CodeWarrior and Macintosh Programmer's Workshop. Header files might need modification for other environments. Always use the latest Universal headers containing definitions for Mac OS Classic compilers.

Metrowerks CodeWarrior for Mac OS Classic

To set up your CIN project, use the project stationery in the `cintools:Metrowerks Files:Project Stationery:LabVIEW CIN MWPPC` folder.

The folder contains a template for new CINs with most of the settings you need. Refer to the `Read Me` file in the `Project Stationery` folder for more information.

To create a CIN for Mac OS Classic, you need your source files and `CINLib.ppc.mwerks` in your CodeWarrior project. LabVIEW installs `CINLib.ppc.mwerks` in the `cintools:Metrowerks Files:PPC Libraries` folder.

If you call any routines within LabVIEW, such as `DSSetHandleSize()` or `SetCINArraySize()`, you also need the `labview.export.stub` file. LabVIEW installs `labview.export.stub` in the `cintools:PowerPC Libraries` folder.

If you call any routines from a system shared library, you must add the appropriate shared library interface file to your project.

When building a CIN using CodeWarrior for PPC, you can set many of the preferences to whatever you want. However, other preferences must be set to specific values to correctly create a CIN. If you do not use the project stationery, make sure you set the following preferences in the CodeWarrior **Preferences** dialog box:

- Clear the **Prefix File** (using **MacHeaders** does not work).
- Set **Struct Alignment** to **68K**.
- Clear all the **Entry Point** fields.
- Set **Export Symbols** to **Use .exp file** and place a copy of the file `projectName.exp` from your `cintools:Metrowerks Files:PPC Libraries` folder in the same folder as your CodeWarrior project. Rename `projectName.exp` to *`projectName`*`.exp`, where *`projectName`* is the name of the project file. CodeWarrior looks in this file to determine what symbols your CIN exports. LabVIEW needs these to link to your CIN.

- Set **Project Type** to **Shared Library**. Set the file name to *`cinName`*`.tmp`, where *`cinName`* is the name of your CIN.
- Set **Type** to `.tmp`.
- Set **Creator** to `LVsb`.
- Add your `cintools` folder to the list of access paths.

Select **Project»Make** to build the CIN.

When you successfully build the *`cinName`*`.tmp` file, use the `lvsbutil.app` application to create the *`cinName`*`.lsb` file.

In the **file selection** dialog box, make sure the **For Power PC** box is checked. Select any other options you want for your CIN, and then select your *`cinName`*`.tmp` file. LabVIEW creates *`cinName`*`.lsb` in the same folder as *`cinName`*`.tmp`.

Macintosh Programmer's Workshop

You can use Macintosh Programmer's Workshop (MPW) to build CINs for Mac OS Classic. The following scripts are available for the MPW environment to help you build CINs.

- `CINMake` uses a simplified form of a makefile you provide. You can run it every time you need to rebuild your CIN.
- `LVMakeMake` is similar to the lvmkmf (LabVIEW Make Makefile) script available for building CINs on UNIX. This script builds a skeletal but complete makefile you can then customize and use with the MPW make tool.

  You must have one makefile for each CIN. Name the makefile by appending `.lvm` to the CIN name to indicate that it is a LabVIEW makefile. The makefile should resemble the following pseudocode. Make sure that each `Dir` command ends with the colon character (`:`).
- `name = name` is the Name for the code and 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 LabVIEW subroutine (`.lsb`) file, `name.lsb`.
- `type = type` is the type of external code you want to create. For CINs, use a type of `CIN`.
- `codeDir = codeDir:` is the complete pathname to the folder containing the `.c` file used for the CIN.
- `cinToolsDir = cinToolsDir:` is the complete pathname to the LabVIEW `cintools:MPW` folder.

- `LVMVers = 2` is the version of `CINMake` script reading this `.lvm` file.
- `inclDir = -i inclDir:` is optional and is the complete or partial pathname to a folder containing any additional `.h` files.
- `otherPPCObjFiles = otherPPCObjFiles` is optional and is the list of additional object files, files with a `.o` extension, your code needs to compile. Separate the names of files with spaces.
- `ShLibs = sharedLibraryNames` is optional and is a 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. Separate the names with spaces.
- `ShLibMaps = sharedLibMappings` is optional and is 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 usually look similar to the following example code:

  ```
  -librename libA.xcoff=libA
  -librename libB.xcoff=libB
  ```

  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`, as shown in the following example code:

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

Go to the MPW Worksheet and enter the following commands, setting your current folder to the CIN folder.

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

Run the LabVIEW `CINMake` script, shown in the following example code:

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

If `CINMake` does not find a `.lvm` file in the current folder, it builds a file named *`cinName`*`.lvm` and prompts you for necessary information. If `CINMake` finds *`cinName`*`.lvm` but it does not have the line `LVMVers = 2`, MPW saves the `.lvm` file in *`cinName`*`.lvm.old` and updates the *`cinName`*`.lvm` file to be compatible with the new version of `CINMake`.

The format of the `CINMake` command is shown in the following example code, with optional parameters listed in brackets:

```
CINMake [-MakeOpts "opts"] [-RShell] [-dbg] [-noDelete]
<name of your CIN>
```

`-MakeOpts`	`opts` specifies extra options to pass to `make`.
`-Rshell`	The `Rshell` option to `CINMake` causes the make commands to execute in ToolServer rather than in the MPW Shell. Having the make commands execute in ToolServer is useful if you want to be able to issue other MPW commands before the make commands have been completed, for example, if the CIN takes a long time to compile.
`-dbg`	If this argument is specified, `CINMake` prints statements describing what it does.
`-noDelete`	If this argument is specified, `CINMake` does not delete temporary files used when making the CIN.

You can use `LVMakeMake` to build an MPW makefile that you can then customize. You should only have to run `LVMakeMake` once for each CIN. You can 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 is shown in the following example code, with optional parameters listed in brackets:

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

`-o`	`makeFile` indicates the name of the output makefile. If this argument is not specified, `LVMakeMake` writes to standard output.

For example, to build a Mac OS Classic makefile for a CIN named `myCIN`, use the following command:

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

You then can 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
```

Load the `.lsb` file that this application creates into your LabVIEW CIN.

Microsoft Windows

To build CINs for LabVIEW for Windows, use the Microsoft Visual C++ or Symantec C compilers.

Visual C++ IDE

Complete the following steps to build CINs using the Visual C++ integrated development environment (IDE).

1. Select **File»New** to create a new DLL project.
2. Select **Win32 Dynamic-Link Library** as the project type. You can name your project whatever you want.
3. Select **An empty DLL project** when prompted to choose the type of DLL that you want to create and click **Finish**.
4. Select **Project»Add To Project»Files** and navigate to your `.c` file. Select your `.c` file and click the **Open** button to add the `.c` file to the project.
5. Select **Project»Add To Project»Files** to add CIN objects and libraries to the project. Select `cin.obj`, `labview.lib`, `lvsb.lib`, and `lvsbmain.def` from the `Cintools` subdirectory. You need these files to build a CIN.
6. Select **Project»Settings** and change **Settings For** to **All Configurations**. Click the **C/C++** tab and set the category to **Preprocessor**. Add the path to your `Cintools` directory in the **Additional include directories** field.
7. Select **Project»Settings** and change **Settings For** to **All Configurations**. Click the **C/C++** tab and set the category to **Code Generation**. Select the **Struct member alignment** tab and select **1 byte**.
8. Choose a run-time library. Select **Project»Settings** and change **Settings for** to **All Configurations**. Select the **C/C++** tab and set the category to **Code Generation**. Select **Multithreaded DLL** in the **Use run-time library** control.

9. Make a custom build command to run `lvsbutil`. Select **Project»Settings** and change **Settings for** to **All Configurations**. Select the **Custom Build** tab and change the **Commands** field as follows, with the code all on a single line:

   ```
   <your path to cintools>\lvsbutil "$(TargetName)" -d
   "$(WkspDir)\$(OutDir)"
   ```

   Change the **Output** fields to `$(OutDir)$(TargetName).lsb`.
10. (For Visual C++.NET compiler only) Add `<CINTOOLSDIR>\lvsbmain.def` to the **Linker»Input»Module Definition File** field.
11. Click the **File View** tab in the **Work Space** window.
12. Open your `.c` file and replace **`/* Insert code here */`** with your code.
13. Select **Build»Build *projectName*.dll**, where ***projectName*** is the name of your project.

Visual C++ Command Line

This section describes using command line tools in Windows to build CINs.

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

   (Windows 2000/NT/XP) You can edit this list with the **System** control panel accessory. For example, if you installed LabVIEW for Windows in `c:\Program Files\National Instruments\LabVIEW` *`x.x`*, the CIN tools directory should be `c:\Program Files\National Instruments\LabVIEW` *`x.x`*`\cintools`, where *`x.x`* is the LabVIEW version number. In this instance, you would add the following line to the user environment variables using the **System** control panel:

   ```
   CINTOOLSDIR = c:\Program Files\National
   Instruments\LabVIEW x.x\cintools
   ```

   (Windows Me/98) Modify your `AUTOEXEC.BAT` to set `CINTOOLSDIR` to the correct value.
2. Build a `.lvm` file (LabVIEW Makefile) for your CIN. You must specify the following items:
   - `name` is the name of your CIN, for example, `mult`.
   - `type` is CIN.
   - `!include $(CINTOOLSDIR)\ntlvsb.mak`

To define additional include paths for a CIN you must add a `CINCLUDES` line to the `.lvm` file, as shown in the following example code:

```
CINCLUDE = -Ipathnames
```

You must include the `-I` argument on the line. `pathnames` is the directory where you look for other includes.

If your CIN uses extra object files, you can specify the `objFiles` option. 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 location of the compiler.

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 libraries, define the symbol `cinLibraries`. For example, to use standard C functions in the previous 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 with spaces.

Symantec C

Building CINs using Symantec C is similar to building CINs for Visual C++ Command Line. However, you should use `smake` instead of `nmake` on your `.lvm` file.

Solaris 2.*x*

LabVIEW for Solaris 2.*x* uses external code compiled in a shared library format. To prepare this library for LabVIEW, use the LabVIEW utility `lvsbutil`. `lvsbutil` is in the `cintools` folder.

The default compiler for Solaris is gcc. If gcc is not installed, cc becomes the default compiler for Solaris.

Linux

The only supported compiler for Linux is gcc.

gcc Compiler

Create a makefile using the shell script `lvmkmf` (LabVIEW Make Makefile), which creates a makefile for a given CIN. Use the standard `make` command to make the CIN code. In addition to compiling the CIN, the makefile puts the code in a form LabVIEW can use.

The format for the `lvmkmf` command is shown in the following example code, with optional parameters listed in brackets.

```
lvmkmf [-o Makefile] LVSBName
```

`LVSBName` is the name of the CIN you want to build. If *`LVSBName`* is `foo`, the compiler assumes the source is `foo.c` and names the output file `foo.lsb`.

`-o` is the name of the makefile `lvmkmf` creates. If you do not specify this argument, the makefile name default is `Makefile`.

The created makefile will be similar to the following example code.

Note In the following example code, entries in parentheses correspond to the Solaris cc compiler. Also, replace *xx* in `lv`*`xx`* with the LabVIEW version number, for example, `lv70`.

```
#

# This Makefile was generated automatically by lvmkmf.

#

CC=gcc                  (CC=cc)

LD=gcc                  (LD=ld)

LDFLAGS=-shared         (LDFLAGS=-G)

XFLAGS=-fPIC            (XFLAGS=-K PIC)

CINDIR=/usr/local/lvxx/cintools

CFLAGS=-I$(CINDIR) $(XFLAGS)

CINLIB=$(CINDIR)/libcin.a

MAKEGLUE=$(CINDIR)/makeglueSVR4.awk

AS=as
```

The makefile produced assumes the `cin.o`, `libcin.a`, and `lvsbutil` files are in certain locations. If these assumptions are incorrect, you can edit the makefile to correct the pathnames.

Step 5. Load the CIN Object Code

To load the code resource, right-click the node and select **Load Code Resource** from the shortcut menu. Select the `.lsb` file you created in the *Step 4. Compile the CIN Source Code* section.

LabVIEW loads your object code into memory and links the code to the current front panel or 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 run.

If you modify the source code, you can load the new version of the object code using the **Load Code Resource** shortcut menu item. 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 you can call from CINs. 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 the following categories:

- Memory manager
- File manager
- Support manager

External code written using the managers is portable, that is, you can compile it without modification on any platform that supports LabVIEW. This portability has the following advantages:

- The LabVIEW application is built on top of the managers. Except for the managers, the LabVIEW source code is identical across platforms.
- The analysis VIs are built mainly from CINs. The source code for these CINs is the same for all platforms.

Refer to the *Manager Overview* section of Chapter 4, *Programming Issues for CINs*, for more information about the memory manager, the file manager, and the support manager.

Refer to Chapter 6, *Function Descriptions*, for descriptions of functions or file manager data structures.

Note When you call the LabVIEW manager functions from a DLL, use `#include extcode.h` in any files that use manager functions and link to `labview.lib`. Set the structure alignment of the compiler to 1 byte. Some of the manager functions, such as `SetCINArraySize`, are CIN-specific, and you cannot call them from a DLL.

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 **ph** parameter `AZCopyHandle`/`DSCopyHandle` allocating and releasing function, or are type defined as such, such as the **name** parameter of the `FNamePtr` function. In most cases, the manager function writes 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 do not have to pre-allocate memory for a return value.

The following functions have parameters that return a value for which you must pre-allocate memory.

`AZMemStats`	`FNamePtr`
`AZCopyHandle/DSCopyHandle`	`FNewRefNum`
`DateToSecs`	`FPathToArr`
`DSMemStats`	`FPathToAZString`
`FCreate`	`FPathToDString`
`FCreateAlways`	`FPathToPath`
`FFlattenPath`	`FRefNumToFD`
`FGetAccessRights`	`FStringToPath`
`FGetEOF`	`FTextToPath`
`FGetInfo`	`FUnflattenPath`
`FGetPathType`	`GetAlong`
`FMOpen`	`SetALong`
`FMRead`	`RandomGen`

`FMTell`	`SecsToDate`
`FMWrite`	`NumericArrayResize`

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

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 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` writes its results there, with unpredictable consequences. Similarly, `FMOpen` writes its results to the space to which `fd` points, which is not a valid place because `fd` is uninitialized.

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);
    }
```

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

Debugging External Code

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

To create this debugging window, use the `DbgPrintf` function. The format for `DbgPrintf` is similar to the format of the `SPrintf` function, described in Chapter 6, *Function Descriptions*. `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 to 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
                          */
```

Windows

Windows supports source-level debugging of CINs using Microsoft's Visual C environment.

Complete the following steps to debug CINs in Windows.

1. Modify your CIN to set a debugger trap. You must do this to force Visual C to load your debugging symbols. The trap call must be done after the CIN is in memory. The easiest way to do this is to place it in the `CINLoad` procedure. After the debugging symbols are loaded,

you can set normal debug points inside Visual C. Windows Me/98 has a single method of setting a debugger trap, while Windows 2000/NT/XP can use the Windows Me/98 method or another.

The method common to Windows is to insert a debugger break using an in-line assembly command, as shown in the following code:

```
_asm int 3;
```

Adding this to `CINLoad` gives you the following code:

```
CIN MgErr CINLoad(RsrcFile reserved)
{
    Unused(reserved);
    _asm int 3;
    return noErr;
}
```

When the debugger trap is hit, Visual C++ invokes a debug window highlighting that line.

In Windows 2000/NT/XP, you can use the `DebugBreak` function. The `DebugBreak` function exists in Windows Me/98 but does not produce suitable results for debugging CINs. To use `DebugBreak`, include `<windows.h>` at the top of your file and place the call where you want to break, as shown in the following example code:

```
#include <windows.h>
CIN MgErr CINLoad(RsrcFile reserved)
{
    Unused(reserved);
    DebugBreak();
    return noErr;
}
```

When that line runs, you will be in assembly. Step out of that function to get to the point of the `DebugBreak` call.

2. Rebuild your CIN with debugging symbols.

   If you built your CIN from the command line, add the following lines to the `.lvm` file of your CIN to add debug information to the CIN:

   ```
   DEBUG = 1
   cinLibraries = Kernel32.lib
   ```

If you built your CIN using the IDE, build a debug version of the DLL. Select **Projects»Settings**, the **Debug** tab, and the **General** category. Enter your LabVIEW executable in **Executable for debug session**.

3. Run LabVIEW.

   If you built your CIN from the command line, start LabVIEW normally. When the debugger trap is run, a the following message appears:

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

   Click the **Cancel** button to launch the debugger, which attaches to LabVIEW, searches for the DLLs, then asks for the source file of your CIN. Point it to your source file, and the debugger loads the CIN source code. You then can debug your code.

   If you built your CIN using the IDE, open your CIN project and click the **GO** button. Visual C launches LabVIEW.

UNIX

On UNIX, you can use standard C `printf` calls or the `DbgPrintf` function described in the previous section. You also can use gdb, the Gnu debugger, to debug the CIN. You must load the VI that contains the CIN before you add breakpoints. The CIN is not loaded until the VI is loaded.

4 Programming Issues for CINs

This chapter describes the data structures LabVIEW uses when passing data to a CIN and describes the function libraries, or managers, that you can use in external code modules. The function libraries include the memory manager, the file manager, and the support manager.

Passing Parameters

LabVIEW passes parameters to the `CINRun` routine. The parameters correspond to each of the wires connected to the CIN. You can pass any data type to a CIN you can construct in LabVIEW. CINs do not have a limit to the number of parameters you can pass to and from them.

Parameters in the CIN .c File

When you right-click a CIN on a block diagram and select **Create .c File** from the shortcut menu, LabVIEW creates a `.c` file in which you can enter your CIN code. The `CINRun` function and its prototype are given. `CINRun` function parameters correspond to the data types being passed to the CIN on the block diagram.

The `.c` file created is a standard C file, except LabVIEW gives the data types unambiguous names. C does not define the size of low-level data types, for example, 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. In the `cintools` directory, LabVIEW includes a header file, `extcode.h`, that contains type definitions that associate LabVIEW data types with the corresponding data type for the supported compilers of each platform.

`extcode.h` defines some constants and types whose definitions might conflict with the definitions of system header files. The `cintools` directory also contains `hosttype.h`, which resolves the differences between definitions in `extcode.h` and definitions in system header files. `hosttype.h` also includes many of the common header files for a given platform

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

If you write a CIN that accepts a single 32-bit signed integer, the `.c` file indicates the `CINRun` routine is passed as `int32` by reference. `extcode.h` defines an `int32` to the appropriate data type for the LabVIEW-supported compiler you use. Therefore, you can use the `int32` data type in external code you write.

Passing Fixed-Size Data to CINs

As described in the *Creating a CIN* section of Chapter 3, *CINs*, 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, follow the rules for each kind of terminal in the *Creating a CIN* section of Chapter 3, *CINs*. 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. The last terminal pair corresponds to the last parameter.

Scalar Numerics

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

Scalar Booleans

LabVIEW stores Boolean data types in memory as 8-bit integers. If any bit of the integer is 1, the Boolean data type is TRUE. Otherwise, the Boolean data type is FALSE. LabVIEW passes Boolean data types to CINs with the same conventions it uses for numerics.

Note In LabVIEW 4.*x* and earlier, Boolean data types were stored as 16-bit integers. If the high bit of the integer was 1, it was TRUE. Otherwise, the Boolean data type was FALSE.

Refnums

LabVIEW treats a refnum the same way as 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 cluster component is another cluster, LabVIEW stores the component 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 code:

```
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 standard C calling conventions, so the header file declares the word CIN to be equivalent to nothing.

The `MgErr` data type is a LabVIEW data type corresponding to a set of error codes 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 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 containing the subVI. Aborting the running VI 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 describe how to create CINs that work with scalar data types. Refer to Chapter 3, *CINs*, for more information about creating CINs.

Creating a CIN That Multiplies Two Numbers

Complete the following steps to create 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. Place two single-precision numeric controls and one single-precision numeric indicator on the front panel. Wire the node as shown in Figure 4-1. **A*B** is wired to an output-only terminal pair.

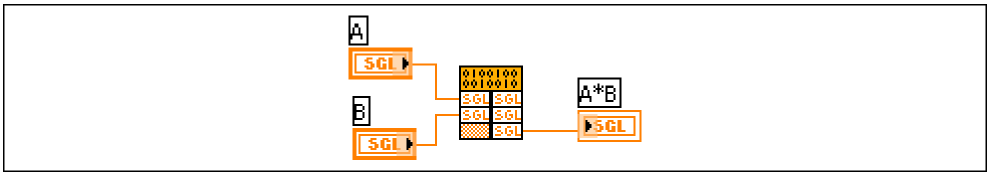


Figure 4-1. mult.vi Block Diagram

4. Save the VI as `mult.vi`.
5. Right-click the CIN and select **Create .c File**. LabVIEW prompts you to select a name and a storage location for a `.c` file.
6. Name the file `mult.c`. LabVIEW creates the following `.c` file:

```
/*
 * CIN source file
 */
#include "extcode.h"
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;
   }
```

The preceding `.c` file contains a prototype and a template for the `CINRun` routine of the CIN. 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 they are wired to the CIN, that is, top to bottom. Thus, `A`, `B`, and `A_B` are pointers to **A**, **B**, and **A*B**, respectively.

As described in the *Parameters in the CIN .c File* section, the `float32` data type is not a standard C data type. For most C compilers, the `float32` data type corresponds to the `float` data type. However, this might 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 you can access. Refer to the *Manager Overview* section of this chapter for descriptions of these data types and routines.

7. Fill in the code for the `CINRun` routine for this multiplication example. You do not have to use the variable names LabVIEW gives you in `CINRun`. You can change the variable names to increase the readability of the code. Replace `/* ENTER YOUR CODE HERE */` in the `.c` file with the following example 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 LabVIEW knows whether the CIN encountered any fatal problems and handles the error correctly.

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

8. Compile the source code and convert it into a form LabVIEW can use. The following sections summarize the steps for each of the supported compilers. Refer to the *Step 4. Compile the CIN Source Code* section of Chapter 3, *CINs*, for more information about completing this step on your platform.

   (Macintosh Programmer's Workshop) Create a file named `mult.lvm`. Make sure the name variable is set to `mult`. Build `mult.lvm`.

   (Metrowerks CodeWarrior) Create a new project and place `mult.c` in it. Build `mult.lsb`.

   (Microsoft Visual C++ Compiler Command Line and Symantec C for Windows) Create a file named `mult.lvm`. Make sure the name variable is set to `mult`. Build `mult.lvm`.

   (Microsoft Visual C++ Compiler IDE for Windows) Create a project.

   (UNIX Compilers) Create a makefile using the shell script `lvmkmf` in the `cintools` directory. For this example, enter the following command:

   ```
   lvmkmf mult
   ```

   The preceding command creates a file called `Makefile`. After running `lvmkmf`, enter the standard `make` command, which uses `Makefile` to create a file called `mult.lsb`. You can load `mult.lsb` into the CIN in LabVIEW.

9. Right-click the node and select **Load Code Resource**. Select `mult.lsb`, the object code file you created.

You should be able to run the VI. If you save the VI, LabVIEW saves the CIN object code along with the VI.

Comparing Two Numbers, Producing a Boolean Scalar

This example shows only the block diagram and the code. Complete the following steps to create a CIN that compares two single-precision numbers.

1. Follow the instructions in the *Creating a CIN* section of Chapter 3, *CINs*, to create the CIN. Figure 4-2 shows the block diagram for this CIN.

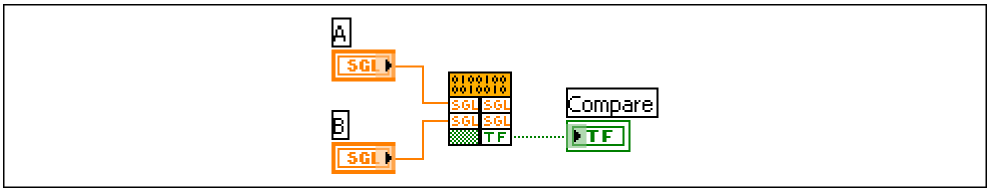


Figure 4-2. aequalb.vi Block Diagram

2. Save the VI as `aequalb.vi`.
3. Create a `.c` file for the CIN and name it `aequalb.c`. LabVIEW creates the following `.c` file:

```
/*
 * CIN source file
 */
#include "extcode.h"
CIN MgErr CINRun(float32 *A, float32 *B,
LVBoolean *compare);
CIN MgErr CINRun(float32 *A, float32 *B,
LVBoolean *compare) {
   if (*A == *B)
      *compare = LVTRUE;
   else
      *compare= LVFALSE;
   return noErr;
   }
```

If the first number is greater than the second one, the return value is TRUE. Otherwise, the return value is FALSE.

Passing Variably-Sized Data to CINs

LabVIEW dynamically allocates memory for arrays and strings. If a string or array needs more space to hold new data, its current location might not offer enough contiguous space to hold the resulting string or array. LabVIEW might have to move the data to a location that offers more space.

To accommodate the 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. Refer to the *Using Pointers and Handles in Memory Zones* section of this chapter for information about handles. Refer to Chapter 6, *Function Descriptions*, for descriptions of specific handle functions.

Alignment Considerations

When a CIN returns variably sized data, you need to adjust the size of the handle that references the array. You can adjust the handle size using the memory manager routine `DSSetHandleSize`. If the data is stored in the application zone, you can use the `AZSetHandleSize` routine to adjust the handle size. Both the `DSSetHandleSize` routine and `the AZSetHandleSize` routine work. However, it is difficult to calculate the size correctly in a platform-independent manner because some platforms have special requirements about how you align and pad memory.

Instead of using `XXSetHandleSize`, 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. Refer to the *Resizing Arrays and Strings* section of this chapter for information about resizing arrays.

The following examples highlight alignment differences for various platforms.

- In Windows, a one-dimensional array of double-precision, floating-point numbers is stored in a handle. 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.

  In Solaris and Mac OS X, 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 makes sure the array data falls on 8-byte boundaries.
- In a three-dimensional array of clusters, each cluster contains a double-precision, floating-point number and a 4-byte integer. As in the previous example, Solaris 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 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 makes sure the next element is properly aligned.

You can use `SetCINArraySize` and `NumericArrayResize` to solve the preceding problems. Refer to Chapter 6, *Function Descriptions*, for information about the `SetCINArraySize` and `NumericArrayResize` functions.

Arrays and Strings

LabVIEW passes arrays by handle, as described in the *Alignment Considerations* section. For an *n*-dimensional array, the handle begins with *n* 4-byte values describing 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. 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 and Boolean arrays in memory as one-dimensional arrays of 8-bit unsigned integers. Refer to the *Using the Flatten To String Function* section of Chapter 1, *Introduction*, for information about using the Flatten to String function to convert LabVIEW data into a string.

Note LabVIEW 4.*x* stored Boolean arrays in memory as a series of bits packed to the nearest 16-bit integer. LabVIEW 4.*x* ignored unused bits in the last 16-bit integer. LabVIEW 4.*x* ordered 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 preceded Boolean arrays. The dimension size for LabVIEW 4.*x* Boolean arrays indicates the number of valid bits contained in the array.

Paths

The exact structure for `Path` data types is subject to change in future versions of LabVIEW. A `Path` is a dynamic data structure LabVIEW passes the same way it passes arrays. LabVIEW stores the data for `Paths` in an application zone handle. Refer to Chapter 6, *Function Descriptions*, for information about the functions that manipulate `Paths`.

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 the component 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

To resize return arrays and strings you pass to a CIN, use the LabVIEW `SetCINArraySize` routine. Pass to the `SetCINArraySize` function the handle you want to resize, information describing the data structure, and the desired size of the array or handle. `SetCINArraySize` takes into account any padding and alignment needed for the data structure. However, `SetCINArraySize` does not 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`. Pass to the `NumericArrayResize` function the array you want to resize, a description of the data structure, and information about the new size of the array.

Consider the following issues when you resize arrays of variably-sized data, such as arrays of strings, with the `SetCINArraySize` or `NumericArrayResize` routines.

- If the new size of the array is smaller, LabVIEW disposes of the handles used by the disposed element. Neither `SetCINArraySize` nor `NumericArrayResize` sets the dimension field of the array. You must set the dimension field of the array in your code after the function call.
- If the new size of the array is larger, LabVIEW does not automatically create the handles for the new elements. You have to create these handles after the function returns.

Refer to Chapter 6, *Function Descriptions*, for more information about the `SetCINArraySize` and `NumericArrayResize` routines.

Examples with Variably Sized Data

The following examples describe how to create CINs that work with variably-sized data types. Refer to Chapter 3, *CINs*, for more information about creating CINs.

Concatenating Two Strings

In this example, the CIN concatenates two strings and uses an input-output terminal. The top left terminal of the CIN takes in the first string as an input parameter to the CIN. The top right terminal of the CIN returns the result of the concatenation. This example shows only the diagram and the code.

Complete the following steps to create the CIN.

1. To create the CIN, follow the instructions in the *Creating a CIN* section of Chapter 3, *CINs*. Figure 4-3 shows the block diagram for this CIN.

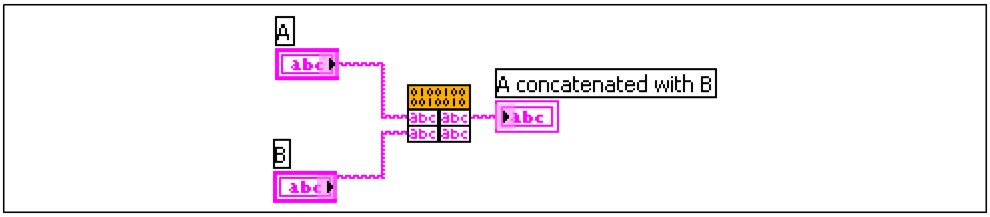


Figure 4-3. lstrcat.vi Block Diagram

2. Save the VI as `lstrcat.vi`.
3. Create a `.c` file for the CIN and name it `lstrcat.c`. LabVIEW creates the following `.c` file:

```
/*
 * CIN source file
 */
#include "extcode.h"
CIN MgErr CINRun(
        LStrHandle var1,
        LStrHandle var2);
CIN MgErr CINRun(
        LStrHandle var1,
        LStrHandle var2) {
    /* ENTER YOUR CODE HERE */
    return noErr;
    }
```

4. Fill in the `CINRun` function with the following example code:

```
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, `NumericArrayResize` resizes `strh1` to hold the additional data.

If `NumericArrayResize` fails, it returns a non-zero value of type `MgErr`. In this example, `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 might free up enough memory so LabVIEW can continue running.

After resizing the string handle, `MoveBlock` copies the second string to the end of the first string. `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

In this example, the CIN accepts two two-dimensional arrays and 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. The Boolean parameter 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.

Complete the following steps to create the CIN.

1. Follow the instructions in the *Creating a CIN* section of Chapter 3, *CINs*, to create the CIN. Figure 4-4 shows the front panel for this VI.

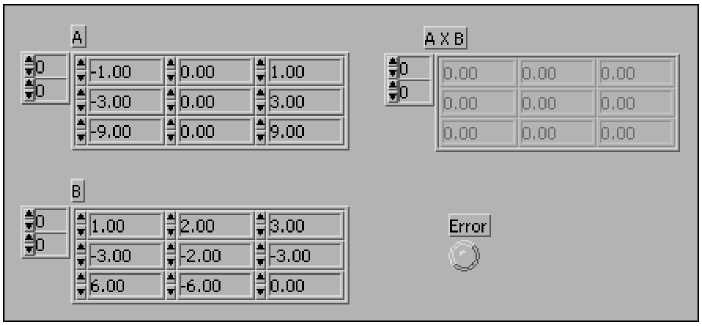


Figure 4-4. cross.vi Front Panel

Figure 4-5 shows the block diagram for this VI.

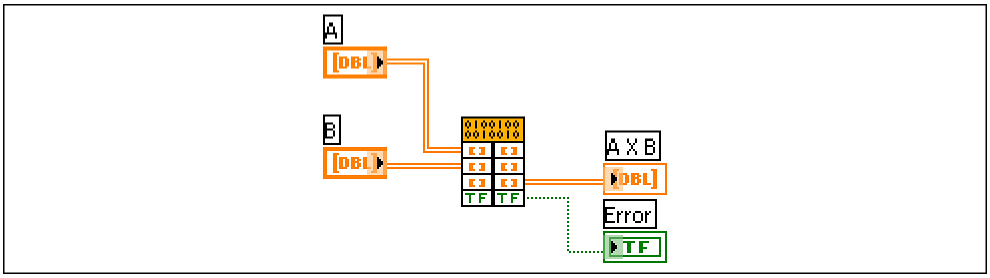


Figure 4-5. cross.vi Block Diagram

2. Save the VI as `cross.vi`.
3. Create the `.c` file for the CIN and save it as `cross.c`. The following code 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 */
/*
```

```
 * typedefs
 */
typedef struct {
   int32 dimSizes[2];
   float64 arg1[1];
   } TD1;
typedef TD1 **TD1Hdl;
CIN MgErr CINRun(TD1Hdl A, TD1Hdl B, TD1HdlAxB, LVBoolean *error);
CIN MgErr CINRun(TD1Hdl A, TD1Hdl B, TD1Hdl AxB, LVBoolean *error){
   int32       i,j,k,l;
   int32       rows, cols;
   float64     *aElmtp, *bElmtp, *resultElmtp;
   MgErr       err = noErr;
   int32       newNumElmts;
   if ((k = (*A)->dimSizes[1]) !=(*B)->dimSizes[0]) {
      *error = LVTRUE;
      goto out;
      }
   *error = LVFALSE;
   rows = (*A)->dimSizes[0];
      /* number of rows in A and result */
   cols = (*B)->dimSizes[1];
      /* number of cols in B and result */
   newNumElmts = rows * cols;
   if (err = SetCINArraySize((UHandle)AxB,
         ParamNumber, newNumElmts))
      goto out;
   (*AxB)->dimSizes[0] = rows;
   (*AxB)->dimSizes[1] = cols;
   aElmtp = (*A)->arg1;
   bElmtp = (*B)->arg1;
   resultElmtp = (*AxB)->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*cols + j];
         resultElmtp++;
         }
out:
   return err;
   }
```

In this example, `CINRun` is the only routine performing substantial operations. `CINRun` cross-multiplies the two-dimensional arrays `A` and `B`. LabVIEW stores the resulting array in `resulth`. If the number of columns in `A` is not equal to the number of rows in `B`, `CINRun` sets `*error` to `LVTRUE` to inform the calling diagram of invalid data.

`SetCINArraySize`, the LabVIEW routine that accounts for alignment and padding requirements, resizes the array. The two-dimensional array data structure is the same as the one-dimensional array data structure, except 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 Figure 4-6.

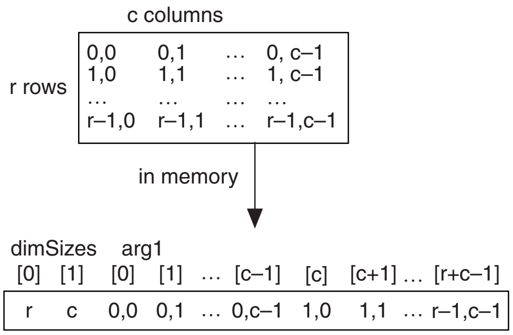


Figure 4-6. LabVIEW Data Storage

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:

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

Working with Clusters

In this example, the CIN takes an array of clusters and a single cluster as inputs. The clusters contain a 16-bit signed integer and a string.The top terminal of the CIN is an input-output terminal. The top terminal takes the array of clusters as an input and returns the new array of clusters as an output.

In addition to the new array of clusters, the CIN returns a Boolean parameter and a 32-bit signed integer. If the single cluster is already present in the array of clusters, the CIN sets the Boolean parameter to TRUE. If the Boolean parameter is TRUE, the CIN returns in the 32-bit integer output the position the single cluster occupies in the array of clusters.

If the single cluster is not present in the array of clusters, the CIN adds it to the array, sets the Boolean output to FALSE, and returns through the 32-bit integer output the position the single cluster now occupies in the new array of clusters.

This example shows only the front panel, block diagram, and source code. Complete the following steps to create the CIN.

1. Follow the instructions in the *Creating a CIN* section of Chapter 3, *CINs*, to create the CIN. Figure 4-7 shows the front panel for this VI.

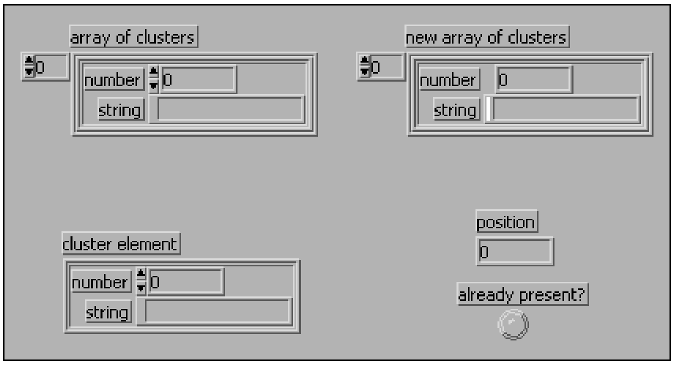


Figure 4-7. tblsrch.vi Front Panel

Figure 4-8 shows the block diagram for this VI.

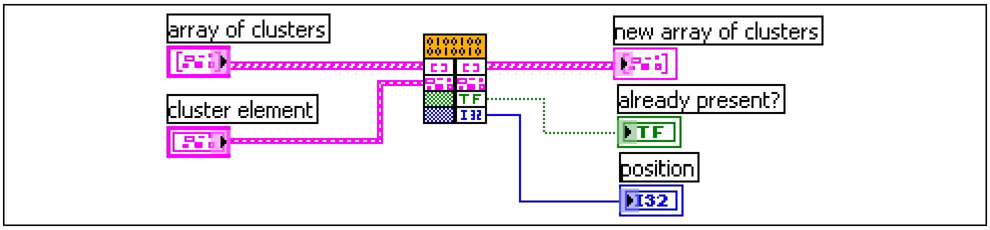


Figure 4-8. tblsrch.vi Block Diagram

2. Save the VI as `tblsrch.vi`.

3. Create the `.c` file and save it as `tblsrch.c`. The following code 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 */
```

```
/*
 * 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;
```

```
            }
            /* DSCopyHandle will allocate a new handle since this is
            NULL */
        newStringh = NULL;
        if(err = DSCopyHandle(&newStringh,elementp->string))
            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 performing substantial operations. `CINRun` first searches through the table to see if the single cluster is present. `CINRun` then compares string components using the LabVIEW routine `LStrCmp`. If `CINRun` finds the single cluster, the routine returns the position the single cluster occupies in the array of clusters.

If the routine does not find the single cluster in the array of clusters, the CIN adds a new element to the array of clusters. The memory manager routine `DSCopyHandle` creates a new handle containing the same string as the one in the single cluster you passed to the CIN. `CINRun` increases the size of the array of clusters using `SetCINArraySize` and fills the last position in the new array of clusters with a copy of the single cluster 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.

Refer to Chapter 6, *Function Descriptions*, for information about `CINRun`, `LStrCmp`, `DSCopyHandle`, and `SetCINArraySize`.

Manager Overview

LabVIEW has a large number of external functions that 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 the manager routines, you can create external code that works on all platforms that LabVIEW supports. LabVIEW contains the following manager functions:

- Memory manager
- File manager
- Support manager

To achieve platform independence, data types should not depend on the peculiarities of various compilers. For example, the C language does not define the size of an integer. Without an explicit definition of the size of each data type, you have difficulty creating code that works identically across multiple compilers. LabVIEW managers use data types that explicitly indicate the size of the data type. 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 you write external code, use the manager data types instead of the host computer data types so your code is more portable and has fewer errors.

The remainder of this chapter discusses data types and the individual manager functions.

Data Types

Manager data types include the following data types:

- Scalar
- Char
- Dynamic
- Memory-related
- Constants

Scalar

Scalar data types include Boolean and numeric.

Boolean

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

LabVIEW block diagrams store Boolean scalars as 8-bit values. The value is 0 if FALSE and 1 if TRUE. This Boolean form is called an `LVBoolean`.

The Table 4-1 describes the two forms of Boolean scalars.

Table 4-1. Boolean Scalars

Name	Description
`Bool32`	32-bit integer, 0 if FALSE, 1 if TRUE
`LVBoolean`	8-bit integer, 0 if FALSE, 1 if TRUE

Numeric

The managers support 8-, 16-, and 32-bit signed and unsigned integers. For floating-point numbers, LabVIEW supports the single, double, and extended floating-point data types.

Table 4-2. Floating-Point Data Types Supported by LabVIEW

Type	Description
Single	32-bit
Double	64-bit
Extended	At least 80-bit

LabVIEW supports complex numbers containing two floating-point numbers, with different complex numeric types for each of the floating-point data types.

LabVIEW supports the following basic data types for numbers:

- Signed integers

– `int8`	8-bit integer
– `int16`	16-bit integer
– `int32`	32-bit integer

- Unsigned integers

– `uInt8`	8-bit integer
– `uInt16`	16-bit integer
– `uInt32`	32-bit integer

- Floating-point numbers

– `float32`	32-bit floating-point number
– `float64`	64-bit floating-point number
– `floatExt`	extended-precision floating-point number

Table 4-3 explains how various platforms store extended-precision numbers.

Table 4-3. Storage Format for Extended-Precision Numbers on Various Platforms

Platform	Storage Format
Windows	80-bit structure with two `int32` components, `mhi` and `mlo`, and an `int16` component, `e`
Mac OS	128-bit double-double format
Sun	128-bit floating-point numbers
Concurrent	Same as `float64`

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. Table 4-4 contains the code for the type definitions for each of these complex data types.

Table 4-4. Code for Complex Numbers

Complex Number Type	Code

32-bit	`typedef struct {`
	`    float32 re, im;`
	`    } cmplx64;`

64-bit	`typedef struct {`
	`    float64 re, im;`
	`    } cmplx128;`

Extended-precision	`typedef struct {`
	`    floatExt re, im;`
	`    } cmplxExt;`

char

The `char` data type is defined by C to be an 8-bit signed integer. LabVIEW defines an unsigned `char` data type, with the following type definition:

```
typedef uInt8 uChar;
```

Dynamic

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

Arrays

LabVIEW supports arrays of any of the basic data types described in this section. 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. Refer to the *Passing Parameters* section of this chapter for examples of manipulating arrays.

Strings

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

C-Style Strings (CStr)

A C-style 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 is the type definition for a C string.

```
typedef uChar *CStr;
```

Pascal-Style Strings (PStr)

A Pascal-style string (`PStr`) is a series of unsigned characters. The value of the first character indicates the length of the string. A `PStr` can have a range of 0 to 255 characters. The following code is the type definition for a Pascal 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. The specified number of characters follow the length of the string. `LStr` is the string data type used by LabVIEW block diagrams. The following code is the type definition for a `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. However, arrays of strings can place a burden on the memory manager because of the large number of dynamic objects it must manage. To make working with lists more efficient, LabVIEW supports the concatenated Pascal string (`CPStr`) data type, which is a list of Pascal-style strings concatenated into a single block of memory. Using the `CPStr` data structure, you can use support manager routines to create and manipulate lists.

The following code is the type definition for a `CPStr` string.

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

Paths

A path (pathname) indicates the location of a file or directory in a file system. LabVIEW has a separate 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 can create and manipulate `Path` data types using file manager routines.

Memory-Related

LabVIEW uses pointers and handles to reference dynamically allocated memory. The memory-related data types have the following type definitions:

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

Refer to Chapter 6, *Function Descriptions*, for information about the use of memory-related data types with functions.

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 (uInt8)
LVTRUE 1 (uInt8)
```

Memory Manager

Most applications need routines for allocating and deallocating memory on request. The memory manager is a set of platform-independent routines you can use to dynamically allocate, manipulate, and deallocate memory. The LabVIEW memory manager supports dynamic allocation of both nonrelocatable and relocatable blocks, using pointers and handles.

If you need to perform dynamic memory allocation or manipulation from external code modules, use the memory manager. If your external code operates on data types other than scalars, you should understand how LabVIEW manages memory and know which utilities manipulate data.

The following code shows how the memory manager defines generic handle and pointer data types.

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

Memory Allocation

Applications use the following types of memory allocation:

- Static
- Dynamic

Note You can allocate memory using `malloc` inside a CIN. However, assign the pointer that results from the `malloc` call to a variable that is local to the CIN code rather than to a variable passed from the LabVIEW block diagram. Use the LabVIEW memory manager functions if you want to create or resize memory associated with a variable passed from the LabVIEW block diagram.

Static Memory Allocation

With static memory allocation, the compiler determines memory requirements when you create an application. When you launch the application, LabVIEW creates memory for the known global memory requirements of the application. The memory LabVIEW creates remains allocated while the application runs. Static memory allocation is simple to work with because the compiler handles all the details.

However, static memory allocation cannot address the memory management requirements of most real-world applications because you cannot determine most memory requirements until run time. Also, statically declared memory might result in larger memory requirements because the memory is allocated for the duration of the application.

Dynamic Memory Allocation

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 than static memory allocation because you have to determine memory requirements and allocate and deallocate memory as necessary.

The LabVIEW memory manager supports the following methods of dynamic memory allocation:

- Using pointers to allocate memory
- Using handles to allocate memory

Using Pointers for Dynamic Memory Allocation

The more conventional method uses pointers to allocate memory. With pointers, you request a block of memory of a certain size. The routine returns the address of the block of memory to your CIN. When you no longer need the block of memory, you call a routine to free the block of memory. You can use the block of memory to store data. You reference the data stored in the block of memory by using the address the manager routine returned when you created the pointer. You can make copies of the pointer and use them in multiple places in your application to refer to the same data.

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

Using Handles for Dynamic Memory Allocation

A second form of memory allocation uses handles. As with pointers, when you allocate memory using handles, you request a block of memory of a certain 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. If you look up the correct address using the handle, you access the correct data.

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

Memory Zones

LabVIEW's memory manager interface can distinguish between the following distinct sections, called zones:

- Data space (DS)
- Application zone (AZ)

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`. Routines for the application zone begin with `AZ`.

Currently, the two zones are actually one zone, but this might change in future releases of LabVIEW. Therefore, you should write applications 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, 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`, which uses AZ handles. You should only use file manager functions, not the AZ memory manager routines, to manipulate `Path`. 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 a handle created using the `SizeHandle` function to a CIN, your CIN should use AZ routines to work with the handle.

Using Pointers and Handles in Memory Zones

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

Create a handle using *`XX`*`NewHandle`, with which you specify the size of the memory block. Create a pointer using *`XX`*`NewPtr`. *`XX`*`NewHandleClr` and *`XX`*`NewPClr` 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 *`XX`*`DisposeHandle` or *`XX`*`DisposePtr`.

If you need to resize an existing handle, use the *`XX`*`SetHandleSize` routine, which 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 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.

While operating within a single call of a CIN node, an AZ handle does not move unless you specifically resize it. In this context, you do not 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 might be relocated between calls. `AZHLock` and `AZHUnlock` might be useful if you do not want the handle to relocate. A DS handle moves only when you resize it.

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 of Using Pointers and Handles in Memory Zones

This simple example demonstrates how to work with pointers and handles in C.

The following code shows how to 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 32-bit signed integer. The first line does not actually allocate memory for the `int32`. The first line 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 the variable is a pointer.

The second line creates a block of memory in the data space large enough to hold a single 32-bit signed 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.

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 a 32-bit signed integer. Strictly speaking, the first line declares `myInt32H` as a pointer to a pointer to an `int32`. As with the previous example, the first line does not allocate memory for the `int32`. The first line 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 the variable is a handle.

The second line creates a block of memory in the data space 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, the second 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.

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 you can use with pointers and for more information about how to work with pointers.

File Manager

The file manager supports routines for opening and creating files, reading data from and writing data to files, and closing files. In addition, 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. You also 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 which you can create and delete directories. You also can 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 also can lock a range of the file, if you need to make sure a range is nonvolatile at a given time.

The file manager also provides many routines for manipulating paths, or path names, in a platform-independent manner. The file manager supports the creation of path descriptions, which are either relative to a specific location or absolute, that is, 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.

Applications that manipulate files can use the functions in the file manager. The file manager routines support basic file operations such as creating, opening, and closing files, writing data to files, and reading data from files. In addition, you can use file manager routines to create directories, determine characteristics of files and directories, and copy files. File manager routines use a LabVIEW data type for file path names, called `Paths`, that indicates a file or directory path independent of the platform. You can translate a `Path` to and from the conventional format a host platform uses for describing a file pathname. Refer to the *File Manager* section of this chapter for more information about the file manager.

Identifying Files and Directories

When you perform operations on files and directories, you need to identify the target of the operation. The platforms LabVIEW supports use a hierarchical file system, meaning files are stored in directories, possibly nested several levels deep. These hierarchical 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 hierarchical file systems, you must include the volume name to specify the location of a file. On other systems, such as UNIX, you do not need to 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, specify the target using the path of the target. The path describes the volume containing the target, the directories between the top level and the target, and the name of the target. If the target is an open file, use a file descriptor to specify that LabVIEW should perform an operation on the open file. The file descriptor is an identifier 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

LabVIEW uses the following types of file path specifications:

- Conventional
- Empty
- LabVIEW specifications

Conventional

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. The last name is the file or directory the path identifies.

A path can be one of the following types:

- Relative path
- Absolute path

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 the path should go up a level from the current location.

(UNIX) 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. Indicate 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
```

The following paths are two relative paths to the same file.

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

(Windows) 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. Indicate 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
```

The following paths are two relative paths to the same file.

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

(Mac OS) 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, that is, it specifies a volume, the path must end with a colon. If the path is a relative path, it begins with a colon. The colon is optional for a relative path consisting of only one name. Indicate 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`

The following paths are two relative paths to the same file.

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

Empty

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.

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

(Windows) You represent the empty absolute path as an empty string. The empty absolute path specifies the set of all volumes on the system. A period (`.`) represents the empty relative path.

(Mac OS) The empty absolute path is represented as an empty string. The absolute path specifies the set of all volumes on the system. A colon (`:`) represents the empty relative path.

LabVIEW

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 a `Path`. 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 a `Path`, the file manager provides routines for comparing, duplicating, determining `Path` characteristics, and converting `Path` 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 open the file again, the new file descriptor is most likely different from the previous file descriptor.

File Refnums

In the file manager, LabVIEW accesses open files using file descriptors. However, on the front panel and block diagram, LabVIEW accesses open files using 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. To pass references to open files into or out of a CIN, convert file refnums to file descriptors and convert file descriptors to file refnums, using the functions described in Chapter 6, *Function Descriptions*.

Support Manager

The support manager contains a collection of constants, macros, basic data types, and functions, such for bit or byte manipulation of data, string manipulation, mathematical operations, sorting, searching, and determining the current time and date.

The 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, Pascal strings, and C strings. Table 4-5 describes the different string types.

Table 4-5. String Types and Their Descriptions

String Type	Description
LabVIEW	4-byte length field followed by data, generally stored in a handle
Pascal	1-byte length field followed by data
C	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 also contains transcendental functions for many complex and extended floating-point operations.

Certain routines specify time as a data structure. The following example code illustrates specifying time as a data structure.

```
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;
```

5 Advanced Applications

This chapter describes several options needed only in advanced applications, including how to use the `CINInit`, `CINDispose`, `CINAbort`, `CINLoad`, `CINUnload`, `CINSave`, and `CINProperties` routines. This chapter also describes how global data works within CIN source code and how Windows users can call a DLL from a CIN.

CIN Routines

A CIN consists of several routines. The routines are described by the `.c` file LabVIEW creates when you right-click the CIN on the block diagram and select **Create .c File** from the shortcut menu. Previous chapters have described only the `CINRun` routine. Other routines include `CINLoad`, `CINInit`, `CINAbort`, `CINSave`, `CINDispose`, `CINUnload`, and `CINProperties`.

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

If you want to preallocate or initialize global state information, you need to understand more of how LabVIEW manages data and CINs, as described in the following sections.

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 CIN. After the object code is loaded into the CIN, LabVIEW loads a copy of the code resource into memory and attaches it to the CIN. When you save the VI, the code resource is saved along with the VI as an attached component. Because the 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 for that VI. A data space is a block of data storage memory. LabVIEW uses the data space to store information such as the values in shift registers. If the VI is reentrant, LabVIEW allocates a data space for each usage of the VI. Refer to the *LabVIEW Help* for more information about reentrancy and other execution properties.

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

When you create a CIN, LabVIEW allocates a CIN data space strictly for the use of the CIN. A CIN data space is a 4-byte storage location in the VI data space(s). Each CIN can have one or more CIN data spaces reserved for the CIN, depending on how many times the CIN 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. Figure 5-1 shows a simple example of data storage spaces for one CIN.

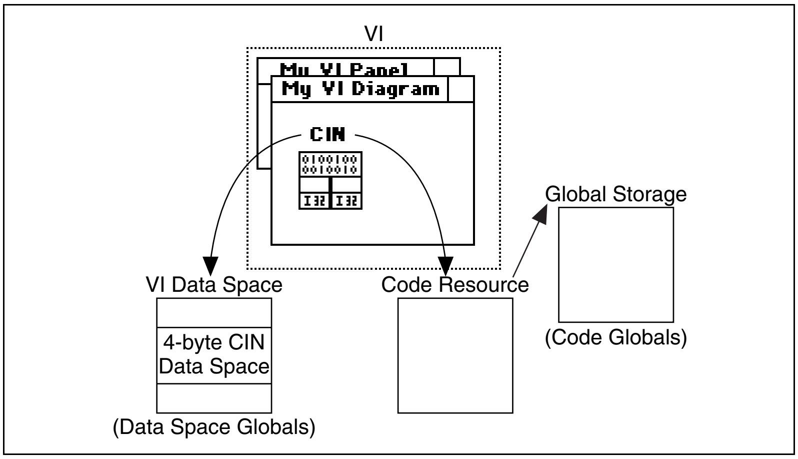


Figure 5-1. Data Storage Spaces for One CIN

A CIN 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 a code resource with the desired name is already loaded into memory, LabVIEW links the CIN to that code resource for execution purposes.

Linking the CIN to the code resource 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 having the same name, only one code resource is actually loaded into memory. Both references to the code resources point to the same code. LabVIEW can verify that a subVI call matches the subVI connector pane terminal. However, LabVIEW cannot verify that your source code matches the CIN call.

One Reference to the CIN in a Single VI

The following section describes the standard case in which you have a code resource referenced by only one CIN and the VI containing the CIN is not reentrant. Other cases have slightly more complicated behavior and are 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. By LabVIEW calling the `CINLoad` routines when you first load a VI, you have a chance to load any file-based resources at load time because LabVIEW calls this routine only when the VI is first loaded. Refer to the *Loading a New Resource into the CIN* section of this chapter for an exception to this rule. After LabVIEW calls the `CINLoad` routine, it calls `CINInit`. Together, `CINLoad` and `CINInit` 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. Because LabVIEW calls `CINLoad` once for a given code resource, 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 corresponding to that data space. If you want to use CIN data space globals, initialize them in `CINInit`. Refer to the *Loading a New Resource into the CIN*, *Compiling a VI* and the *Code Globals and CIN Data Space Globals* sections of this chapter for more information about `CINInit` and data space globals.

Unloading a VI

When you close a VI front panel, LabVIEW checks whether any references to the VI are in memory. If any references to the VI are in memory, 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, the 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 *XX*`NewHandle` in your `CINInit` routine, you should use *XX*`DisposeHandle` 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 allocated in `CINLoad`. As with `CINDispose` and `CINInit`, `CINUnload` is called for each `CINLoad`. For example, if you loaded some resources from a file in `CINLoad`, you can free the memory 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. LabVIEW calls `CINDispose` for each CIN data space and each reference to the code resource, followed by the `CINUnload` for the old resource.

After the calls to `CINDispose` and `CINUnload`, the new code resource is given a chance to perform any initialization it needs to perform. LabVIEW calls `CINLoad` for the new code resource. After LabVIEW calls `CINLoad`, it calls the `CINInit` routine 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, including resetting all uninitialized shift registers to their default values. Also, your CIN is given a chance to dispose or initialize any storage it manages.

LabVIEW completes the following steps when compiling a VI and before disposing of the current data space.

1. Calls the `CINDispose` routine for each reference to the code resource within the VI(s) being compiled. Calling the `CINDispose` routine gives the code resource a chance to dispose of any old results it is managing.
2. Compiles the VI and creates a new data space for the VI(s) being compiled. LabVIEW creates multiple data spaces for reentrant VIs.
3. Calls `CINInit` for each reference to the code resource within the compiled VI(s). Calling `CINInit` gives the code resource a chance to create or initialize any data it wants to manage.

Running a VI

Click the **Run** button in a VI to run the VI. 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. Calling the `CINSave` routine gives you the chance to save any resources, such as something you loaded in `CINLoad`. 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, you need to save in `CINSave` anything you expect to be able to load in `CINLoad`.

Aborting a VI

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

Note `CINAbort` only works if the VI containing the CIN is running. If a top level VI is running and the program is aborted, `CINAbort` only works if the top level VI is currently running the subVI in which the CIN is located. If the top level VI is running another subVI in the hierarchy, `CINAbort` does not work.

CINs are synchronous. Therefore, when a CIN begins execution, the CIN takes control of its thread until execution is completed. If your version of LabVIEW is single-threaded, you cannot abort the CIN because no other LabVIEW tasks can run while a CIN executes.

Multiple References to the Same CIN in a Single VI

If you loaded the same code resource into multiple CINs, or if you duplicated a given CIN, 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, LabVIEW calls the `CINLoad` routine only once when the resource is first loaded into memory, although it is also called if you load a new version of the resource. When you unload the VI, LabVIEW calls `CINUnload` once.

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

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.

Figure 5-2 shows an example of three CINs referencing the same code resource.

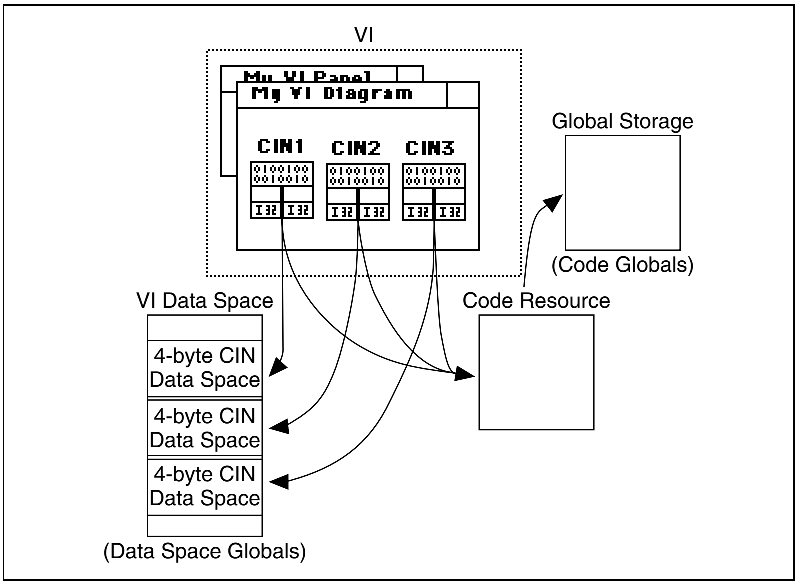


Figure 5-2. Three CINs Referencing the Same Code Resource

Multiple References to the Same CIN in Different VIs

Making multiple references to the same CIN in different VIs is different for single-threaded operating systems than for mutlithreaded operating systems.

Note Mac Classic is the only single-threaded operating system supported by LabVIEW

Single-Threaded Operating Systems

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

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

If you have a reentrant VI containing 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 5-3 shows an example of three VIs referencing a reentrant VI containing one CIN.

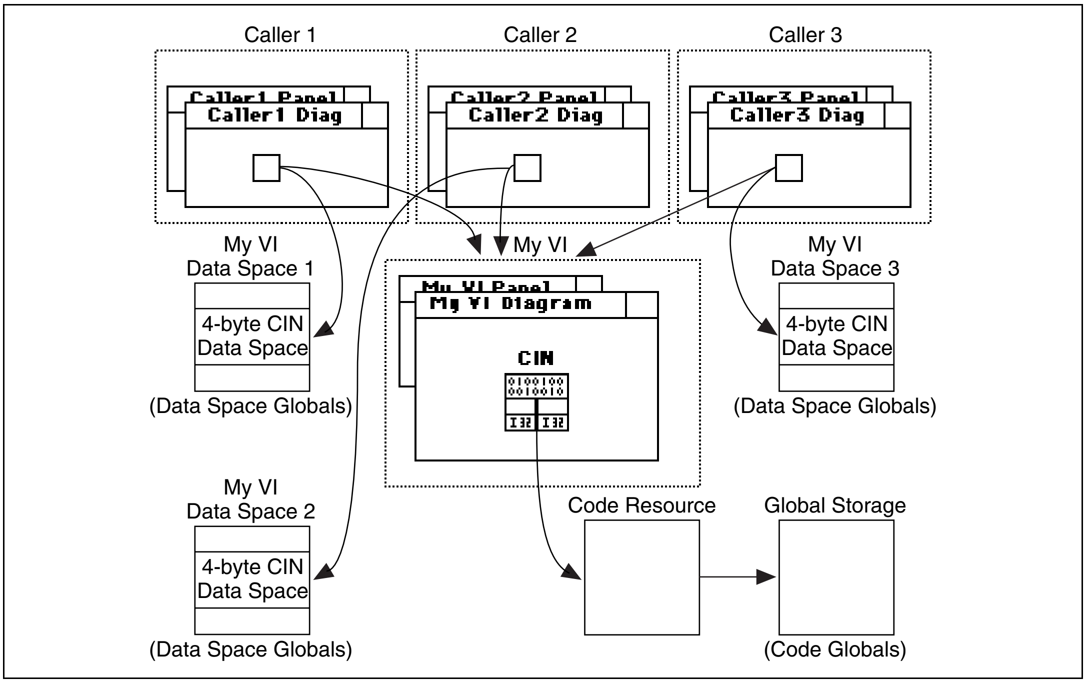


Figure 5-3. Three VIs Referencing a Reentrant VI Containing One CIN

Multithreaded Operating Systems

By default, CINs written in LabVIEW 5.0 or earlier run in a single thread, the user interface thread. When you change a CIN to be reentrant, that is, to run in multiple threads, more than one execution thread can call the CIN at the same time. Add the following code to your `.c` file if you want a CIN to run in the current execution thread of the block diagram.

```
CIN MgErr CINProperties(int32 mode, void *data)
    {

    switch (mode) {
        case kCINIsReentrant:
            *(Bool32 *)data = TRUE;
            return noErr;
            break;
        }

    return mgNotSupported;
    }
```

If you read and write a global or static variable, or call a non-reentrant function within your CINs, keep the execution of those CINs in a single thread. Even if a CIN is marked reentrant, the CIN functions other than `CINRun` are called from the user interface thread. For example, `CINInit` and `CINDispose` are never called from two different threads at the same time. `CINRun` might be running when the user interface thread is calling `CINInit`, `CINAbort`, or any of the other functions.

To be reentrant, the CIN must be safe to call `CINRun` from multiple threads and safe to call any of the other `CIN` procedures and `CINRun` at the same time. Other than `CINRun`, you do not need to protect any of the `CIN` procedures from each other because calls to them are always in one thread.

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 the global, regardless of whether the VI is reentrant or whether you have multiple references to the same code resource in memory.

In some cases, you might 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 CIN. 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 also can call `SetDSStorage` to set the value of the CIN data space for this instance. You can use the storage location set by `SetDSStorage` to store any 4-byte quantity you want to have for each instance of one of these globals. If you need more than four bytes of global data, store a handle or pointer to a structure containing your globals.

The following code examples show the exact syntax of the `GetDSStorage` and `SetDSStorage` routines defined in `extcode.h`.

- `int32 GetDSStorage(void);`

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

Code Globals and CIN Data Space Globals Examples

The following 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, as shown in Figure 5-4.

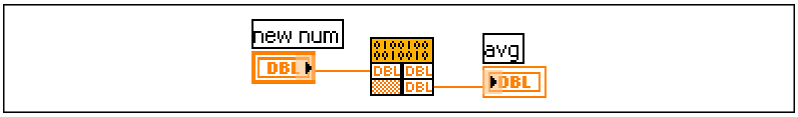


Figure 5-4. Averaging CIN

When you write your application, 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 CINs or different reentrant VIs affects the same set of globals. In the code globals averaging example, the result indicates the average of all values passed to the CIN.

If you use CIN data space globals, each CIN calling the same code resource and each VI can have its own set of globals, if the VI is reentrant. In the CIN data space globals averaging example, the results 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 containing that CIN is not reentrant, choose either method.

Using Code Globals

The following code averages using code globals.

```
/*
 * CIN source file
 */

#include "extcode.h"

float64 gTotal;
int32 gNumElements;

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;
    }
```

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 allocate and deallocate the memory using `CINLoad` and `CINUnload` because `CINLoad` and `CINUnload` are called only once, regardless of the number of references to the code resources and the number of data spaces. This example does not use the `UseDefaultCINLoad` macro because this `.c` file has a `CINLoad` function.

Using CIN Data Space Globals

The following code uses CIN data space globals to perform the averaging.

```
/*
 * CIN source file
 */

#include "extcode.h"

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 err;
   }

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;
    }
```

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 makes sure `GetDSStorage` and `SetDSStorage` return the 4-byte CIN data space value for that node or CIN data space. When you want to access the data in the CIN data space, use `GetDSStorage` to retrieve the handle and then dereference the appropriate fields. Finally, use the `CINDispose` routine you need to dispose of the handle.

6 Function Descriptions

This chapter describes the CIN functions you can use with LabVIEW. You can use these functions 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 CIN manager routines are platform-independent, so you can create CINs that work on all platforms supported by LabVIEW.

Refer to the *Manager Overview* section of Chapter 4, *Programming Issues for CINs*, for general information about the manager routines.

Memory Manager Functions

The memory manager functions can dynamically allocate, manipulate, and release memory.

You can perform the following operations by using the functions listed.

- Handles and pointers, verifying
  - `AZCheckHandle/DSCheckHandle`
  - `AZCheckPtr/DSCheckPtr`
- Handles, allocating and releasing
  - `AZCopyHandle/DSCopyHandle`
  - `SetCINArraySize`
  - `NumericArrayResize`
  - `AZDispose Handle/DSDisposeHandle`
  - `AZGetHandleSize/DSGetHandleSize`
  - `AZNewHandle/DSNewHandle`
  - `AZNewHClr/DSNewHClr`
  - `AZRecoverHandle/DSRecoverHandle`
  - `AZSetHandleSize/DSSetHandleSize`
  - `AZSetHSzClr/DSSetHSzClr`

- Handles, manipulating properties
  - `AZHLock`
  - `AZHPurge`
  - `AZHNoPurge`
  - `AZHUnlock`
- Memory utilities
  - `ClearMem`
  - `MoveBlock`
  - `SwapBlock`
- Memory zone utilities
  - `AZHeapCheck/DSHeapCheck`
  - `AZMaxMem/DSMaxMem`
  - `AZMemStats/DSMemStats`
- Pointers, allocating and releasing
  - `AZDisposePtr/DSDisposePtr`
  - `AZNewPClr/DSNewPClr`
  - `AZNewPtr/DSNewPtr`

File Manager Functions

The file manager functions can create, open and close files, write data to files, and read data from files. In addition, file manager routines can create directories, determine characteristics of files and directories, and copy files.

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. Use file manager routines to create and manipulate the `Path` data type.

Permissions for Files and Directories

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`.

(UNIX) The nine bits of permissions correspond exactly to nine UNIX permission bits governing read, write, and execute permissions for user, group, and others. Figure 6-1 shows permission bits in UNIX.

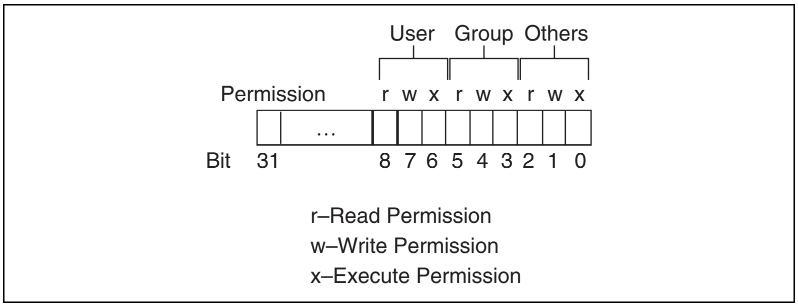


Figure 6-1. Permission Bits in UNIX

(Windows) 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.

(Mac OS) All nine bits are used for directories (folders). The bits which control read, write, and execute permissions, respectively, in UNIX are used to control See Files, Make Changes, and See Folders access rights, respectively, in Mac OS.

File Manager Functions and Operations

You can perform the following operations by using the functions listed.

- Current position mark, positioning
  - `FMSeek`
  - `FMTell`
- Default access rights information, getting
  - `FGetDefGroup`
- Directory contents, creating and determining
  - `FListDir`
  - `FNewDir`
- End-of-file mark, positioning
  - `FGetEOF`
  - `FSetEOF`
- File data to disk, flushing
  - `FFlush`

- File operations, performing basic
  - `FCreate`
  - `FCreateAlways`
  - `FMClose`
  - `FMOpen`
  - `FMRead`
  - `FMWrite`
- File range, locking
  - `FLockOrUnlockRange`
- File refnums, manipulating
  - `FDisposeRefNum`
  - `FIsARefNum`
  - `FNewRefNum`
  - `FRefNumToFD`
  - `FRefNumToPath`
- File, directory, and volume information determination
  - `FExists`
  - `FGetAccessRights`
  - `FGetInfo`
  - `FGetVolInfo`
  - `FSetAccessRights`
  - `FSetInfo`
- Filenames and patterns, matching
  - `FStrFitsPat`
- Files and directories, moving and deleting
  - `FMove`
  - `FRemove`
- Files, copying
  - `FCopy`
- Path type, determining
  - `FGetPathType`
  - `FIsAPathOfType`
  - `FSetPathType`

- Path, extracting information
  - `FDepth`
  - `FDirName`
  - `FName`
  - `FNamePtr`
  - `FVolName`
- Paths, comparing
  - `FIsAPath`
  - `FIsAPathOrNotAPath`
  - `FIsEmptyPath`
  - `FPathCmp`
- Paths, converting to and from other representations
  - `FArrToPath`
  - `FFlattenPath`
  - `FPathToArr`
  - `FPathToAZString`
  - `FPathToDSString`
  - `FStringToPath`
  - `FTextToPath`
  - `FUnFlattenPath`
- Paths, creating
  - `FAddPath`
  - `FAppendName`
  - `FAppPath`
  - `FEmptyPath`
  - `FMakePath`
  - `FNotAPath`
  - `FRelPath`
- Paths, disposing
  - `FDestroyPath`
- Paths, duplicating
  - `FPathCpy`
  - `FPathToPath`

Support Manager Functions

You can use the support manager functions for bit or byte manipulation of data, string manipulation, mathematical operations, sorting, searching, and determining the current time and date.

You can perform the following operations by using the functions listed.

- Byte manipulation operations
  - `Cat4Chrs`
  - `GetALong`
  - `Hi16`
  - `HiByte`
  - `HiNibble`
  - `Lo16`
  - `LoByte`
  - `Long`
  - `LoNibble`
  - `Offset`
  - `SetALong`
  - `Word`
- Mathematical operations
  - `Abs`
  - `Max`
  - `Min`
  - `Pin`
  - `RandomGen`
- String manipulation
  - `BlockCmp`
  - `CPStrBuf`
  - `CPStrCmp`
  - `CPStrIndex`
  - `CPStrInsert`
  - `CPStrLen`
  - `CPStrRemove`

- `CPStrReplace`
- `CPStrSize`
- `CToPStr`
- `FileNameCmp`
- `FileNameIndCmp`
- `FileNameNCmp`
- `FPrintf`
- `HexChar`
- `IsAlpha`
- `IsDigit`
- `IsLower`
- `IsUpper`
- `LStrBuf`
- `LStrCmp`
- `LStrLen`
- `LStrPrintf`
- `LToPStr`
- `PPrintf`
- `PPrintfp`
- `PPStrCaseCmp`
- `PPStrCmp`
- `PStrBuf`
- `PStrCaseCmp`
- `PStrCat`
- `PStrCmp`
- `PStrCpy`
- `PStrLen`
- `PStrNCpy`
- `PToCStr`
- `PToLStr`
- `SPrintF`
- `SPrintfp`
- `StrCat`

  - `StrCmp`
  - `StrCpy`
  - `StrLen`
  - `StrNCaseCmp`
  - `StrNCmp`
  - `StrNCpy`
  - `ToLower`
  - `ToUpper`
- Synchronization functions
  - `Occur`
  - `PostLVUserEvent`
- Utility functions
  - `BinSearch`
  - `QSort`
  - Unused
- Time functions
  - `ASCIITime`
  - `DateCString`
  - `DateToSecs`
  - `MilliSecs`
  - `SecsToDate`
  - `TimeCString`
  - `TimeInSecs`

Mathematical Operations

In addition to the mathematical operations in the preceding list, LabVIEW supports a number of other mathematical functions. The following functions are implemented as defined in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie.

```
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);
```

Individual Function Descriptions

The remainder of this chapter describes individual CIN functions you can use with LabVIEW.

Abs

```
int32 Abs(n);
```

Purpose

Returns the absolute value of **n**, unless **n** is -2^{31}, in which case the function returns the number unmodified.

Parameters

Name	Type	Description
n	`int32`	`int32` whose absolute value you want to determine.

ASCIITime

```
CStr ASCIITime(secs);
```

Purpose

Returns a pointer to a string representing the date and time of day corresponding to *t* seconds after January 1, 1904, 12:00 a.m., UT. 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. The time is in the same format as that returned by the `TimeCString` function using a mode of `0`. For example, this function might return **`Tuesday, Dec 22, 1992 5:30`**. In SPARCstation, this function accounts for international conventions for representing dates.

Parameters

Name	Type	Description
secs	`uInt32`	Seconds since January 1, 1904, 12:00 a.m., UT.

Return Value

The date and time as a C string.

AZCheckHandle/DSCheckHandle

```
MgErr AZCheckHandle(h);
MgErr DSCheckHandle(h);
```

Purpose

Verifies that the specified handle is a handle. If it is not a handle, this function returns `mZoneErr`.

Parameters

Name	Type	Description
h	`Uhandle`	Handle you want to verify.

Return Value

`MgErr`, which can contain the following errors:

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

AZCheckPtr/DSCheckPtr

```
MgErr AZCheckPtr(p);
MgErr DSCheckPtr(p);
```

Purpose

Verifies that the specified pointer is allocated with `XXNewPtr` or `XXNewPClr`. If it is not a pointer, this function returns `mZoneErr`.

Parameters

Name	Type	Description
p	`UPtr`	Pointer you want to verify.

Return Value

`MgErr`, which can contain the following errors:

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

AZCopyHandle/DSCopyHandle

```
MgErr AZCopyHandle(void *ph, const void *hsrc)
MgErr DSCopyHandle(void *ph, const void *hsrc)
```

Purpose

Copies the data referenced by the handle **hsrc** into the handle pointed to by **ph** or a new handle if **ph** points to `NULL`.

Parameters

Name	Type	Description
ph	`UHandle*`	Pointer to the handle to copy the data into. This must point to a valid handle or `NULL`. If it points to `NULL`, a new handle is allocated.
hsrc	`UHandle`	The handle containing the data to copy.

Return Value

`MgErr`, which can contain the following errors:

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

AZDisposeHandle/DSDisposeHandle

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

Purpose

Releases the memory referenced by the specified handle.

Parameters

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

Return Value

`MgErr`, which can contain the following errors:

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

AZDisposePtr/DSDisposePtr

```
MgErr AZDisposePtr(p);
MgErr DSDisposePtr(p);
```

Purpose

Releases the memory referenced by the specified pointer.

Parameters

Name	Type	Description
p	`UPtr`	Pointer you want to dispose of.

Return Value

`MgErr`, which can contain the following errors:

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

AZGetHandleSize/DSGetHandleSize

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

Purpose

Returns the size of the block of memory referenced by the specified handle.

Parameters

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

Return Value

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

AZHeapCheck/DSHeapCheck

```
int32 AZHeapCheck(Bool32 d);
int32 DSHeapCheck(Bool32 d);
```

Purpose

Verifies that the specified heap is not corrupt. This function returns **`0`** for an intact heap and a nonzero value for a corrupt heap.

Parameters

Name	Type	Description
d	`Bool32`	Heap you want to verify.

Return Value

`int32`, which can contain the following errors:

`noErr`	The heap is intact.
`mCorruptErr`	The heap is corrupt.

AZHLock

`MgErr AZHLock(h);`

Purpose

Locks the memory referenced by the application zone handle **h** so 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. Locking handles interferes with efficient memory management. Also, do not enlarge a locked handle.

Parameters

Name	Type	Description
h	`UHandle`	Application zone handle you want to lock.

Return Value

`MgErr`, which can contain the following errors:

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

AZHNoPurge

```
void AZHNoPurge(h);
```

Purpose

Marks the memory referenced by the application zone handle **h** as not purgative.

Parameters

Name	Type	Description
h	`UHandle`	Application zone handle you want to mark as not purgative.

AZHPurge

```
void AZHPurge(h);
```

Purpose

Marks the memory referenced by the application zone handle **h** as purgative. 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 purgative, check the handle before using it to determine whether it has become an empty handle.

Parameters

Name	Type	Description
h	`UHandle`	Application zone handle you want to mark as purgative.

AZHUnlock

`MgErr AZHUnlock(h);`

Purpose

Unlocks the memory referenced by the application zone handle **h** so 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.

Parameters

Name	Type	Description
h	`UHandle`	Application zone handle you want to unlock.

Return Value

`MgErr`, which can contain the following errors:

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

AZMaxMem/DSMaxMem

```
int32 AZMaxMem();
int32 DSMaxMem();
```

Purpose

Returns the size of the largest block of contiguous memory available for allocation.

Return Value

`int32`, the size of the largest block of contiguous memory available for allocation.

AZMemStats/DSMemStats

```
void AZMemStats(MemStatRec *msrp);
void DSMemStats(MemStatRec *msrp);
```

Purpose

Returns various statistics about the memory in a zone.

Parameters

Name	Type	Description
msrp	`MemStatRec`	Statistics about the zone's free memory in a MemStatRec structure. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

The following code defines the `MemStatRec` structure:

```
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 the contiguous memory blocks. **maxFreeSize** is the largest of the contiguous memory blocks, as returned by `XXMaxMem`. **nFreeBlocks** is the number of the contiguous memory 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 the contiguous memory blocks. **maxAllocSize** is the largest of the contiguous memory 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 the values of **nPointers**, **nUnlockedHdls**, and **nLockedHdls** together to find the total number of allocated blocks.

The four reserved fields are reserved for use by National Instruments.

AZNewHandle/DSNewHandle

```
UHandle AZNewHandle(size);
UHandle DSNewHandle(size);
```

Purpose

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.

Parameters

Name	Type	Description
size	`int32`	Size, in bytes, of the handle you want to create.

Return Value

A handle of the specified size. If an error occurs, this function returns `NULL`.

AZNewHClr/DSNewHClr

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

Purpose

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

Parameters

Name	Type	Description
size	`int32`	Size, in bytes, of the handle you want to create.

Return Value

A handle of the specified **size**, where the block of memory is set to all zeros. If an error occurs, this function returns `NULL`.

AZNewPClr/DSNewPClr

```
UPtr AZNewPClr(size);
UPtr DSNewPClr(size);
```

Purpose

Creates a new pointer to a non-relocatable block of memory of the specified **size** and initializes the memory to zero.

Parameters

Name	Type	Description
size	`int32`	Size, in bytes, of the pointer you want to create.

Return Value

A pointer to a block of **size** bytes filled with zeros. If an error occurs, this function returns `NULL`.

AZNewPtr/DSNewPtr

```
UPtr AZNewPtr(size);
UPtr DSNewPtr(size);
```

Purpose

Creates a new pointer to a non-relocatable block of memory of the specified **size**.

Parameters

Name	Type	Description
size	`int32`	Size, in bytes, of the pointer you want to create.

Return Value

A pointer to a block of **size** bytes. If an error occurs, this function returns `NULL`.

AZRecoverHandle/DSRecoverHandle

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

Purpose

Given a pointer to a block of memory that was originally declared as a handle, this function 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.

Parameters

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

Return Value

A handle to the block of memory to which **p** refers. If an error occurs, this function returns `NULL`.

AZSetHandleSize/DSSetHandleSize

```
MgErr AZSetHandleSize(h, size);
MgErr DSSetHandleSize(h, size);
```

Purpose

Changes the size of the block of memory referenced by the specified handle.

While LabVIEW arrays are stored in DS handles, do 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, use either `NumericArrayResize` or `SetCINArraySize`. Refer to the *Resizing Arrays and Strings* section of Chapter 4, *Programming Issues for CINs*, for information about using `NumericArrayResize` and `SetCINArraySize`. Do not use these functions on a locked handle.

Parameters

Name	Type	Description
h	`UHandle`	Handle you want to resize.
size	`int32`	New size, in bytes, of the handle.

Return Value

`MgErr`, which can contain the following errors:

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

AZSetHSzClr/DSSetHSzClr

```
MgErr AZSetHSzClr(h, size);
MgErr DSSetHSzClr(h, size);
```

Purpose

Changes the size of the block of memory referenced by the specified handle and sets any new memory to zero. Do not use this function on a locked handle.

Parameters

Name	Type	Description
h	`UHandle`	Handle you want to resize.
size	`int32`	New size, in bytes, of the handle.

Return Value

`MgErr`, which can contain the following errors:

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

BinSearch

```
int32 BinSearch(arrayp, n, elmtSize, key, compareProcP);
```

Purpose

Searches an array of an arbitrary data type using the binary search algorithm. In addition to passing the array 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);
```

Parameters

Name	Type	Description
arrayp	`UPtr`	Pointer to an array of data.
n	`int32`	Number of elements in the array you want to search.
elmtSize	`int32`	Size in bytes of an array element.
key	`Uptr`	Pointer to the data for which you want to search.
compareProcP	`ProcPtr`	Comparison routine you want `BinSearch` to use to compare array elements. `BinSearch` passes this routine the addresses of two elements that it needs to compare.

Return Value

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 **key** should be placed.

BlockCmp

```
int32 BlockCmp(p1, p2, numBytes);
```

Purpose

Compares two blocks of memory to determine whether one is less than, equal to, or greater than the other.

Parameters

Name	Type	Description
p1	`UPtr`	Pointer to a block of memory.
p2	`UPtr`	Pointer to a block of memory.
numBytes	`int32`	Number of bytes you want to compare.

Return Value

A negative number, zero, or a positive number if **p1** is less than, equal to, or greater than **p2**, respectively.

Cat4Chrs

Macro

```
int32 Cat4Chrs(a,b,c,d);
```

Purpose

Constructs an `int32` parameter from four `uInt8` parameters, with the first parameter as the high byte and the last parameter as the low byte.

Parameters

Name	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`.

Return Value

The resulting `int32`.

ClearMem

```
void ClearMem(p, size);
```

Purpose

Sets **size** bytes starting at the address referenced by **p** to `0`.

Parameters

Name	Type	Description
p	`UPtr`	Pointer to block of memory you want to clear.
size	`int32`	Number of bytes you want to clear.

CPStrBuf

Macro

```
uChar *CPStrBuf(sp);
```

Purpose

Returns the address of the first string in a concatenated list of Pascal strings, that is, the address of `sp->str`.

Parameters

Name	Type	Description
sp	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.

Return Value

The address of the first string of the concatenated list of Pascal strings.

CPStrCmp

```
int32 CPStrCmp(s1p, s2p);
```

Purpose

Lexically compares two concatenated lists of Pascal strings to determine whether one is less than, equal to, or greater than the other. This comparison is case sensitive. The function compares the lists as if they were one string.

Parameters

Name	Type	Description
s1p	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.
s2p	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.

Return Value

`<0`, **`0`**, or **`>0`** if **s1p** is less than, equal to, or greater than **s2p**, respectively. Returns **`<0`** if **s1p** is an initial substring of **s2p**.

CPStrIndex

```
PStr CPStrIndex(s1h, index);
```

Purpose

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, this function returns the pointer to the last string.

Parameters

Name	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
index	`int32`	Number of the string you want, with `0` as the first string.

Return Value

A pointer to the specified Pascal string.

CPStrInsert

`MgErr CPStrInsert(s1h, s2, index);`

Purpose

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, this function places the new string at the end of the list. The function resizes the list to make room for the new string.

Parameters

Name	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
s2	`PStr`	Pointer to a Pascal string.
index	`int32`	Position you want the new Pascal string to have in the list of Pascal strings, with `0` as the first string.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mFullErr`	Insufficient memory.

CPStrLen

Macro

```
int32 CPStrLen(sp);
```

Purpose

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.

Parameters

Name	Type	Description
sp	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.

Return Value

The number of strings in the concatenated list of Pascal strings.

CPStrRemove

```
void CPStrRemove(s1h, index);
```

Purpose

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, this function removes the last string. The function resizes the list after removing the string.

Parameters

Name	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
index	`int32`	Number of the string you want to remove, with `0` as the first string.

CPStrReplace

```
MgErr CPStrReplace(s1h, s2, index);
```

Purpose

Replaces a Pascal string in a concatenated list of Pascal strings with a new Pascal string.

Parameters

Name	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 you want to replace, with `0` as the first string.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mFullErr`	Insufficient memory.

CPStrSize

```
int32 CPStrSize(sp);
```

Purpose

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.

Parameters

Name	Type	Description
sp	`CPStrPtr`	Pointer to a concatenated list of Pascal strings.

Return Value

The number of characters in the concatenated list of Pascal strings.

CToPStr

```
int32 CToPStr(cstr, pstr);
```

Purpose

Converts a C string to a Pascal string, even if the pointers **cstr** and **pstr** refer to the same memory location. If the length of **cstr** is greater than 255 characters, this function converts only the first 255 characters. The function assumes **pstr** is large enough to contain **cstr**.

Parameters

Name	Type	Description
cstr	`CStr`	Pointer to a C string.
pstr	`PStr`	Pointer to a Pascal string.

Return Value

The length of the string, truncated to a maximum of 255 characters.

DateCString

```
CStr DateCString(secs, fmt);
```

Purpose

Returns a pointer to a string representing the date corresponding to **secs** seconds after January 1, 1904, 12:00 a.m., UT. In SPARCstation, this function accounts for international conventions for representing dates.

Note This function was formerly called `DateString`.

Parameters

Name	Type	Description
secs	`uInt32`	Seconds since January 1, 1904, 12:00 a.m., UT.
fmt	`int32`	Indicates the format of the returned date string, using the following values: • `0`—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. For example, `12/31/92`. • `1`—Long date format, *`dayName`*`,` *`MonthName`*`,` *`DayOfMonth`*`,` *`LongYear`*. For example, `Thursday, December 31,1992`. • `2`—Abbreviated date format, *`AbbrevDayName`*`,` *`AbbrevMonthName`*`,` *`DayOfMonth`*`,` *`LongYear`*. For example, `Thu, Dec 31,1992`.

Return Value

The date as a C string.

DateToSecs

```
uint32 DateToSecs(dateRecordP);
```

Purpose

Converts from a time described using the `DateRec` data structure to the number of seconds since January 1, 1904, 12:00 a.m., UT.

Parameters

Name	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**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

The corresponding number of seconds since January 1, 1904, 12:00 a.m., UT.

FAddPath

```
MgErr FAddPath(basePath, relPath, newPath);
```

Purpose

Creates an absolute path by appending a relative path to an absolute path. You can pass the same path variable for the new path that you use for **basePath** or **relPath**. Therefore, you can call this function in the following three ways:

- `err = FAddPath(basePath, relPath, newPath);`

  `/* the new path is returned in a third path variable */`
- `err = FAddPath(path, relPath, path);`

  `/* the new path writes over the old base path */`
- `err = FAddPath(basepath, path, path);`

  `/* the new path writes over the old relative path */`

Parameters

Name	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`.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.

FAppendName

```
MgErr FAppendName(path, name);
```

Purpose

Appends a file or directory name to an existing path.

Parameters

Name	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 you want to append to the existing path.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.

FAppPath

```
MgErr FAppPath(p);
```

Purpose

Indicates the path to the LabVIEW application currently running.

Parameters

Name	Type	Description
p	`Path`	Path in which `FAppPath` stores the path to the current application. **p** must already be an allocated path.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.
`fNotFound`	File not found.
`fIOErr`	Unspecified I/O error.

FArrToPath

`MgErr FArrToPath(arr, relative, path);`

Purpose

Converts a one-dimensional LabVIEW array of strings to a path of the type specified by **relative**. Each string in the 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.

Parameters

Name	Type	Description
arr	`UHandle`	`DS` handle containing the array of strings you want to convert to a path.
relative	`Bool32`	If `TRUE`, 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.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.

FCopy

```
MgErr FCopy(oldPath, newPath);
```

Purpose

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.

Parameters

Name	Type	Description
oldPath	`Path`	Path of the file or directory you want to copy.
newPath	`Path`	Path, including filename, where you want to store the new file.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNotFound`	File not found.
`fNoPerm`	Access was denied; the file, directory, or disk is locked or protected.
`fDiskFull`	Disk is full.
`fDupPath`	The new file already exists.
`fIsOpen`	The original file is open for writing.
`fTMFOpen`	Too many files are open.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error.

FCreate

```
MgErr FCreate(fdp, path, permissions, openMode, denyMode, group);
```

Purpose

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, the function returns an error.

You can use **denyMode** to control concurrent access to the file from within LabVIEW. You can use the **group** parameter to assign the file to a UNIX group. In Windows or Mac OS, **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 you call this function, make sure that you understand how to use the **fdp** parameter. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using the **fdp** parameter.

Parameters

Name	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**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.
path	`Path`	Path of the file you want to create.
permissions	`int32`	Permissions to assign to the new file.
openMode	`int32`	Access mode to use in opening the file. The following values 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. The following values 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.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fIsOpen`	File is already open for writing. This error is returned only in Mac OS and Solaris. Windows returns `fIOErr` when the file is already open for writing.
`fNoPerm`	Access was denied because the file is locked or protected.
`fDupPath`	A file of that name already exists.
`fTMFOpen`	Too many files are open.
`fIOErr`	Unspecified I/O error.

FCreateAlways

```
MgErr FCreateAlways(fdp, path, permissions, openMode, denyMode, group);
```

Purpose

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. You can use the **group** parameter to assign the file to a UNIX group. In Windows or Mac OS, **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 you call this function, make sure that you understand how to use the **fdp** parameter. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using the **fdp** parameter.

Parameters

Name	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**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.
path	`Path`	Path of the file you want to create.
permissions	`int32`	Permissions to assign to the new file.
openMode	`int32`	Access mode to use in opening the file. The following values 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. The following values 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.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fIsOpen`	File is already open for writing. This error is returned only in Mac OS and Solaris. Windows returns `fIOErr` when the file is already open for writing.
`fNoPerm`	Access was denied because the file is locked or protected.
`fDupPath`	A file of that name already exists.
`fTMFOpen`	Too many files are open.
`fIOErr`	Unspecified I/O error.

FDepth

```
int32 FDepth(path);
```

Purpose

Computes the depth, or number of component names, of a path.

Parameters

Name	Type	Description
path	`Path`	Path whose depth you want to determine.

Return Value

`int32`, indicating the depth of the path, which can contain the following values:

`-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.

FDestroyPath

```
void FDestroyPath(&pp);
```

Purpose

Release the memory of an allocated path and `NULL` the path pointer.

Parameters

Name	Type	Description
pp	`Path*`	A pointer to the path you want to deallocate.

Return Value

None.

Note This function replaces the older `FDisposePath` function. The older function is still available, but its use is discouraged. By passing a pointer to the path instead of the path directly, the `FDestroyPath` function can properly `NULL` out the path, thus preventing double deallocation errors. The new function can also handle `NULL` paths, eliminating the need to check for `NULL` prior to calling `FDisposePath`. A typical use of the new function is illustrated by the following text:

```
Path p;
p = FNotAPath(NULL);
// insert code here that uses the path
FDestroyPath(&p);
// p == NULL at this point
```

FDirName

```
MgErr FDirName(path, dir);
```

Purpose

Creates a path for the parent directory of a specified **path**. You can pass the same path variable for the parent path that you use for **path**. Therefore, you can call this function in the following two ways:

- `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 */`

Parameters

Name	Type	Description
path	`Path`	Path whose parent path you want to determine.
dir	`Path`	Parameter in which `FDirName` stores the parent path.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.

FDisposeRefNum

```
MgErr FDisposeRefNum(refNum);
```

Purpose

Disposes of the specified file **refNum**.

Parameters

Name	Type	Description
refNum	`LVRefNum`	File refnum of which you want to dispose.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mgArgErr`	File refnum is not valid.

FEmptyPath

`Path FEmptyPath(p);`

Purpose

Makes an empty absolute path, which is not the same as disposing the path.

Parameters

Name	Type	Description
p	`Path`	Path allocated by `FEmptyPath`. If `NULL`, `FEmptyPath` allocates a new path and returns the value. If **p** is a path, `FEmptyPath` sets the existing path to an empty path and returns the new **p**.

Return Value

The resulting path; if **p** was not `NULL`, the return value is the same empty absolute path as **p**. If an error occurs, this function returns `NULL`.

FExists

```
int32 FExists(path);
```

Purpose

Returns information about the specified file or directory. It returns less information than `FGetInfo`, but it is much quicker on most platforms.

Parameters

Name	Type	Description
path	`Path`	Path of the file or directory about which you want information.

Return Value

`int32`, which can contain the following values:

`kFIsFile`	Specified item is a file.
`kFIsFolder`	Specified item is a directory or folder.
`kFNotExist`	Specified item does not exist.

FFlattenPath

```
int32 FFlattenPath(p, fp);
```

Purpose

Converts **path** into a flat form that you can use to write the path as information to a file. This function stores the resulting flat path in a pre-allocated buffer and returns the number of bytes.

To determine the size needed for the flattened path, pass `NULL` for **fp**. The function returns the necessary size without writing anything into the location pointed to by **fp**.

Parameters

Name	Type	Description
path	`Path`	Path you want to flatten.
fp	`UPtr`	Address in which `FFlattenPath` stores the resulting flattened path. If `NULL`, `FFlattenPath` does not write anything to this address, but does return the size that the flattened path would require. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`int32`, indicating the number of bytes required to store the flattened path.

FFlush

`MgErr FFlush(fd);`

Purpose

Writes any buffered data for the specified file out to the disk.

Parameters

Name	Type	Description
fd	`File`	File descriptor associated with the file.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Not a valid file descriptor.
`fIOErr`	Unspecified I/O error.

FGetAccessRights

```
MgErr FGetAccessRights(path, owner, group, permPtr);
```

Purpose

Returns access rights information about the specified file or directory.

Parameters

Name	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. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNotFound`	File not found.
`fIOErr`	Unspecified I/O error.

FGetDefGroup

```
LStrHandle FGetDefGroup(groupHandle);
```

Purpose

Gets the LabVIEW default group for a file or directory.

Parameters

Name	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.

Return Value

The resulting `LStrHandle`. If **groupHandle** was not `NULL`, the return value is the same `LStrHandle` as **groupHandle**. If an error occurs, this function returns `NULL`.

FGetEOF

```
MgErr FGetEOF(fd, sizep);
```

Purpose

Returns the size of the specified file.

Parameters

Name	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, ***sizep** is undefined. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Not a valid file descriptor.
`fIOErr`	Unspecified I/O error.

FGetInfo

```
MgErr FGetInfo(path, infop);
```

Purpose

Returns information about the specified file or directory.

Parameters

Name	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, **infop** is undefined. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

`FInfoPtr` is a data structure that defines the attributes of a file or directory. The following code lists the file/directory information record, `FInfoPtr`.

```
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;
```

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNotFound`	File not found.
`fIOErr`	Unspecified I/O error.

FGetPathType

`MgErr FGetPathType(path, typePtr)`

Purpose

Returns the type, relative, absolute, or not a path, of a path.

Parameters

Name	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 absolute. • `fRelPath`—The path is relative. • `fNotAPath`—The path is the canonical invalid path or an error occurred. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.

FGetVolInfo

```
MgErr FGetVolInfo(path, vinfo);
```

Purpose

Gets a path specification and information for the volume containing the specified file or directory.

Parameters

Name	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, **path** is undefined.
vinfo	`VInfoRec *`	Address at which `FgetVolInfo` stores the information about the volume. If an error occurs, **vinfo** is undefined. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

The following code describes the volume information record, `VInfoRec`.

```
typedef struct {
                uint32   size;       /* size in bytes of a
                                     volume */
                uint32   used;       /* number of bytes used on
                                     volume */
                uint32   free;       /* number of bytes available
                                     for use on volume */
                }      VInfoRec;
```

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fIOErr`	Unspecified I/O error.

FileNameCmp

Macro

```
int32 FileNameCmp(s1, s2);
```

Purpose

Lexically compares two file names to determine whether one is less than, equal to, or greater than the other. This comparison uses the same case sensitivity as the file system, that is, case-insensitive for Mac OS and Windows, case-sensitive for SPARCstation.

Parameters

Name	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

Return Value

`<0`, **`0`**, or **`>0`** if **s1** is less than, equal to, or greater than **s2**, respectively. Returns **`<0`** if **s1** is an initial substring of **s2**.

FileNameIndCmp

Macro

```
int32 FileNameIndCmp(s1p, s2p);
```

Purpose

Lexically compares two file names and determines whether one is less than, equal to, or greater than the other. This comparison uses the same case sensitivity as the file system, that is, case-insensitive for Mac OS and Windows, case-sensitive for SPARCstation. This function is similar to `FileNameCmp`, except you pass the function handles to the string data instead of pointers.

Parameters

Name	Type	Description
s1p	`PStr *`	Pointer to a Pascal string.
s2p	`PStr *`	Pointer to a Pascal string.

Return Value

`<0`, **`0`**, or **`>0`** if **s1p** is less than, equal to, or greater than **s2p**, respectively. Returns **`<0`** if **s1p** is an initial substring of **s2p**.

FileNameNCmp

Macro

```
int32 FileNameNCmp(s1, s2, n);
```

Purpose

Lexically compares two file names to determine whether one is less than, equal to, 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 Mac OS and Windows, case-sensitive for SPARCstation.

Parameters

Name	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.
n	`uInt32`	Maximum number of characters you want to compare.

Return Value

`<0`, **`0`**, or **`>0`** if **s1** is less than, equal to, or greater than **s2**, respectively. Returns **`<0`** if **s1** is an initial substring of **s2**.

FIsAPath

```
Bool32 FIsAPath(path);
```

Purpose

Determines whether **path** is a valid path.

Parameters

Name	Type	Description
path	`Path`	Path you want to verify.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	Path is well formed and type is absolute or relative.
`FALSE`	Path is not valid.

FIsAPathOfType

```
Bool32 FIsAPathOfType(path, ofType);
```

Purpose

Determines whether a path is a valid path of the specified type, relative or absolute.

Parameters

Name	Type	Description
path	`Path`	Path you want to compare to the specified type.
ofType	`int32`	Type you want to compare to the path's type. **ofType** can have the following values: • `fAbsPath`—Compare the path's type to absolute. • `fRelPath`—Compare the path's type to relative.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	Path is well formed and type is identical to **ofType**.
`FALSE`	Otherwise.

FIsAPathOrNotAPath

```
Bool32 FIsAPathOrNotAPath(path);
```

Purpose

Determines whether **path** is a valid path or the canonical invalid path.

Parameters

Name	Type	Description
path	`Path`	Path you want to verify.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	Path is well formed and type is absolute, relative, or not a path.
`FALSE`	Path is not valid.

FIsARefNum

`Bool32 FIsARefNum(refNum);`

Purpose

Determines whether **refNum** is a valid file refnum.

Parameters

Name	Type	Description
refNum	`LVRefNum`	File refnum you want to verify.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	File refnum has been created and not yet disposed.
`FALSE`	File refnum is not valid.

FIsEmptyPath

```
Bool32 FIsEmptyPath(path);
```

Purpose

Determines whether **path** is a valid empty path.

Parameters

Name	Type	Description
path	`Path`	Path you want to verify.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	Path is well formed and empty and type is absolute or relative.
`FALSE`	Path is not a valid empty path.

FListDir

```
MgErr FListDir(path, list, typeH);
```

Purpose

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. Refer to the *Data Types* section of Chapter 4, *Programming Issues for CINs*, 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**.

Parameters

Name	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 by 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`, `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**.

The following code shows the file type record.

```
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, `kIsFile` 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, `kRecognizedType` is clear. The `kIsLink` bit is set if the item described is a UNIX link or Mac OS alias. Otherwise, `kIsLink` is clear. The `kFIsInvisible` bit is set if the item described does not appear in a file dialog box. Otherwise, `kFIsInvisible` 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 in Mac OS and is computed by examining the file name extension on other systems.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNotFound`	The directory was not found.
`fNoPerm`	Access was denied; the file, directory, or disk is locked or protected.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error.

FLockOrUnlockRange

```
MgErr FLockOrUnlockRange(fd, mode, offset, count, lock);
```

Purpose

Locks or unlocks a section of a file.

Parameters

Name	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, using the following values: • `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`. • `fCurrent`—The first byte to lock or unlock is located **offset** bytes from the current position mark. **offset** can be positive, `0`, or negative. • `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`	Indicates whether `FLockOrUnlockRange` locks or unlocks a range of bytes. If `TRUE` the function locks a range. If `FALSE` the function unlocks a range.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`fIOErr`	Unspecified I/O error.

FMakePath

```
Path FMakePath(path, type, [volume, directory, directory, ..., name,] NULL);
```

The brackets indicate that the **volume**, **directory**, and **name** parameters are optional.

Purpose

Creates a new path. If **path** is `NULL`, this function allocates and returns a new path. Otherwise, **path** is set to the new path, and this function returns **path**. If an error occurs or **path** is not specified correctly, the function returns `NULL`.

When you finish using a path, dispose of it using `FDestroyPath`.

Parameters

Name	Type	Description
path	`Path`	Parameter in which `FMakePath` returns the new path if **path** is not `NULL`.
type	`int32`	Type of path you want to create. If `fAbsPath`, the new path is absolute. If `fRelPath`, the new path is relative.
volume	`PStr`	(Optional) Pascal string containing a legal volume name. An empty string indicates to go up a level in the path hierarchy. This parameter is used only for absolute paths in Mac OS or Windows.
directory	`PStr`	(Optional) Pascal string containing a legal directory name. An empty string indicates to go up a level in the path hierarchy.
name	`PStr`	(Optional) File or directory name. An empty string indicates to go up a level in the path hierarchy.
NULL	`PStr`	Marker indicating the end of the path.

Return Value

The resulting path. If you specified **path**, the return value is the same as **path**. If an error occurs, this function returns `NULL`.

FMClose

```
MgErr FMClose(fd);
```

Purpose

Closes the file associated with the file descriptor **fd**.

Parameters

Name	Type	Description
fd	`File`	File descriptor associated with the file you want to close.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Not a valid file descriptor.
`fIOErr`	Unspecified I/O error.

FMOpen

```
MgErr FMOpen(fdp, path, openMode, denyMode);
```

Purpose

Opens a file with the name and location specified by **path** for writing and reading, as specified by **openMode**.

You can use **denyMode** to control concurrent access to the file from within LabVIEW.

If the function opens 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 you call this function, make sure that you understand how to use the **fdp** parameter. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using the **fdp** parameter.

Parameters

Name	Type	Description
fdp	`File *`	Address at which `FMOpen` stores the file descriptor for the new file. If `FMOpen` fails, it stores `0` in the address **fdp**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.
path	`Path`	Path of the file you want to create.

Name	Type	Description
openMode	`int32`	Access mode to use in opening the file. The following values are defined in the file `extcode.h`. • `openReadOnly`—Open for reading. • `openWriteOnly`—Open for writing; file is not truncated (data is not removed). In Mac OS, this mode provides true write-only access to files. In Windows or UNIX, 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 in Windows or UNIX. • `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. The following values 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.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fIsOpen`	File is already open for writing. This error is returned only in Mac OS and Solaris. Windows returns `fIOErr` when the file is already open for writing.
`fNotFound`	File not found.
`fTMFOpen`	Too many files are open.
`fIOErr`	Unspecified I/O error.

FMove

```
MgErr FMove(oldPath, newPath);
```

Purpose

Moves a file or renames it if the new path indicates the file is to remain in the same directory.

Parameters

Name	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 to move the file or directory.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNotFound`	File not found.
`fNoPerm`	Access was denied; the file, directory, or disk is locked or protected.
`fDiskFull`	Disk is full.
`fDupPath`	The new file already exists.
`fIsOpen`	The original file is open for writing.
`fTMFOpen`	Too many files are open.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error.

FMRead

```
MgErr FMRead(fd, inCount, outCountp, buffer);
```

Purpose

Reads **inCount** bytes from the file specified by the file descriptor **fd**. The function starts from the current position mark and reads the data into memory, starting at the address specified by **buffer**. Refer to the *FMSeek* and *FMTell* functions in this chapter for more information about the current position mark.

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 is zero if any other error occurs.

Parameters

Name	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` does not store any value if `NULL` is passed. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.
buffer	`Uptr`	Address where `FMRead` stores the data.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Not a valid file descriptor or **inCount** < `0`.
`FEOF`	EOF encountered.
`fIOErr`	Unspecified I/O error.

FMSeek

```
MgErr FMSeek(fd, ofst, mode);
```

Purpose

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.

Parameters

Name	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, using the following values: • `fStart`—Current position mark moves to **ofst** bytes relative to the start of the file. **ofst** must be greater than or equal to `0`. • `fCurrent`—Current position mark moves **ofst** bytes from the current position mark. **ofst** can be positive, `0`, or negative. • `fEnd`—Current position mark moves to **ofst** bytes from the end of the file. **ofst** must be less than or equal to `0`.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	The file descriptor is not valid.
`fEOF`	Attempt to seek before the start or after the end of the file.
`fIOErr`	Unspecified I/O error.

FMTell

`MgErr FMTell(fd, ofstp);`

Purpose

Returns the position of the current position mark in the file.

Parameters

Name	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, **ofstp** is undefined. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	The file descriptor is not valid.
`fIOErr`	Unspecified I/O error.

FMWrite

`MgErr FMWrite(fd, inCount, outCountp, buffer);`

Purpose

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. Refer to the *FMSeek* and *FMTell* functions in this chapter for more information about the current position mark.

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 is zero if any other error occurs.

Parameters

Name	Type	Description
fd	`File`	File descriptor associated with the file from 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 written. `FMWrite` does not store any value if `NULL` is passed. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.
buffer	`Uptr`	Address of the data you want to write.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Not a valid file descriptor or **inCount** < `0`.
`fDiskFull`	Out of space.
`fNoPerm`	Access was denied.
`fIOErr`	Unspecified I/O error.

FName

```
MgErr FName(path, name);
```

Purpose

Copies the last component name of a specified path into a string handle and resizes the handle as necessary.

Parameters

Name	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.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Badly formed path or path is root directory.
`mFullErr`	Insufficient memory.

FNamePtr

```
MgErr FNamePtr(path, name);
```

Purpose

Copies the last component name of a path to the address specified by **name**. This routine does not allocate space for the returned data. Therefore, **name** must specify allocated memory of sufficient size to hold the component name.

Parameters

Name	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. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Badly formed path or path is root directory.
`mFullErr`	Insufficient memory.

FNewDir

```
MgErr FNewDir(path, permissions);
```

Purpose

Creates a new directory with the specified **permissions**. If an error occurs, the function does not create the directory.

Parameters

Name	Type	Description
path	`Path`	Path of the directory you want to create.
permissions	`int32`	Permissions for the new directory.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNoPerm`	Access was denied; the file, directory, or disk is locked or protected.
`fDupPath`	Directory already exists.
`fIOErr`	Unspecified I/O error.

FNewRefNum

`MgErr FNewRefNum(path, fd, refNumPtr);`

Purpose

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.

Parameters

Name	Type	Description
path	`Path`	Path of the open file for which you want to create a file refnum.
fd	`File`	File descriptor of the open file for which you want to create a file refnum.
refNumPtr	`LVRefNum *`	Address at which `FNewRefNum` stores the new file refnum. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.

FNotAPath

```
Path FNotAPath(p);
```

Purpose

Creates a path that is the canonical invalid path.

Parameters

Name	Type	Description
p	`Path`	Path allocated by `FNotAPath`. If `NULL`, `FNotAPath` allocates a new canonical invalid path and returns the value. If **p** is a path, `FNotAPath` sets the existing path to the canonical invalid path and returns the new **p**.

Return Value

The resulting path. If **p** is not `NULL`, the return value is the same canonical invalid path as **p**. If an error occurs, this function returns `NULL`.

FPathCmp

```
int32 FPathCmp(lsp1, lsp2);
```

Purpose

Compares two paths.

Parameters

Name	Type	Description
lsp1	`Path`	First path you want to compare.
lsp2	`Path`	Second path you want to compare.

Return Value

`int32`, which can contain the following values:

`-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.

FPathCpy

```
MgErr FPathCpy(dst, src);
```

Purpose

Duplicates the path specified by **src** and stores the resulting path in the existing path, **dst**.

Parameters

Name	Type	Description
dst	`Path`	Path where `FPathCpy` places the resulting duplicate path. This path must already have been created.
src	`Path`	Path you want to duplicate.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.

FPathToArr

```
MgErr FPathToArr(path, relativePtr, arr);
```

Purpose

Converts a **path** to a one-dimensional LabVIEW array of strings and determines whether the **path** is relative. Each component name of the **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.

Parameters

Name	Type	Description
path	`Path`	Path you want to convert to an array of strings.
relativePtr	`Bool32 *`	Address at which to store a Boolean value indicating whether the specified path is relative. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, 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.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Badly formed path or unallocated array.
`mFullErr`	Insufficient memory.

FPathToAZString

`MgErr FPathToAZString(p, txt);`

Purpose

Converts a path to an `LStr` and stores the string as an application zone handle. The `LStr` contains the platform-specific syntax for the path.

Parameters

Name	Type	Description
p	`Path`	Path you want to convert to a string.
txt	`LstrHandle *`	Address at which `FPathToAZString` stores the resulting string. If nonzero, the function assumes it is a valid handle, resizes the handle, fills in its value, and stores the handle at the address referred to by **txt**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error.

FPathToDSString

`MgErr FPathToDSString(p, txt);`

Purpose

Converts a path to an `LStr` and stores the string as a data space zone handle. The `LStr` contains the platform-specific syntax for the path.

Parameters

Name	Type	Description
p	`Path`	Path you want to convert to a string.
txt	`LstrHandle *`	Address at which `FPathToDSString` stores the resulting string. If nonzero, the function assumes it is a valid handle, resizes the handle, fills in its value, and stores the handle at the address referred to by **txt**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error.

FPathToPath

`MgErr FPathToPath(p);`

Purpose

Duplicates a path and returns the new path in the same variable.

Parameters

Name	Type	Description
p	`Path *`	Address of the path you want to duplicate. Variable to which `FPathToPath` returns the resulting path. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.

FRefNumToFD

`MgErr FRefNumToFD(refNum, fdp);`

Purpose

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.

Parameters

Name	Type	Description
refNum	`LVRefNum`	The file refnum whose associated file descriptor you want to get.
fdp	`File *`	Address at which `FRefNumToFD` stores the file descriptor associated with the specified file refnum. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mgArgErr`	File refnum is not valid.

FRefNumToPath

```
MgErr FRefNumToPath(refNum, path);
```

Purpose

Gets the path associated with the specified file refnum and stores the resulting path in the existing **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.

Parameters

Name	Type	Description
refNum	`LVRefNum`	The file refnum whose associated path you want to get.
path	`Path`	Path where `FRefNumToPath` stores the path associated with the specified file refnum. This path must already have been created.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.

FRelPath

```
MgErr FRelPath(startPath, endPath, relPath);
```

Purpose

Computes a relative path between two absolute paths. You can pass the same path variable for the new path that you use for **startPath** or **relPath**. Therefore, you can call this function in the following three ways:

- `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 */`

Parameters

Name	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`.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`mFullErr`	Insufficient memory.

FRemove

```
MgErr FRemove(path);
```

Purpose

Deletes a file or a directory. If an error occurs, this function does not remove the file or directory.

Parameters

Name	Type	Description
path	`Path`	Path of the file or directory you want to delete.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNotFound`	File not found.
`fNoPerm`	Access was denied; the file, directory, or disk is locked or protected.
`fIsOpen`	File is open or directory is not empty.
`fIOErr`	Unspecified I/O error.

FSetAccessRights

`MgErr FSetAccessRights(path, owner, group, permPtr);`

Purpose

Sets access rights information for the specified file or directory. If an error occurs, no information changes.

Parameters

Name	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`.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNotFound`	File not found.
`fIOErr`	Unspecified I/O error.

FSetEOF

`MgErr FSetEOF(fd, size);`

Purpose

Sets the size of the specified file. If an error occurs, the file size does not change.

Parameters

Name	Type	Description
fd	`File`	File descriptor associated with the file.
size	`int32 *`	New file size in bytes.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	Not a valid file descriptor or **size** < `0`.
`fDiskFull`	Disk is full.
`fNoPerm`	Access was denied; the file already exists or the disk is locked or protected.
`fIOErr`	Unspecified I/O error.

FSetInfo

```
MgErr FSetInfo(path, infop);
```

Purpose

Sets information for the specified file or directory. If an error occurs, no information changes.

Parameters

Name	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.

`FInfoPtr` is a data structure that defines the attributes of a file or directory. The following code lists the file/directory information record, `FInfoPtr`.

```
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;
```

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.
`fNotFound`	File not found.
`fIOErr`	Unspecified I/O error.

FSetPathType

```
MgErr FSetPathType(path, type);
```

Purpose

Changes the type of a path, which must be a valid path, to the specified type, relative or absolute.

Parameters

Name	Type	Description
path	`Path`	Path whose type you want to change.
type	`int32`	New type you want the path to have. **type** can have the following values: • `fAbsPath`—The path is absolute. • `fRelPath`—The path is relative.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mgArgErr`	Badly formed path or invalid type.

FStrFitsPat

```
Bool32 FStrFitsPat(pat, str, pLen, sLen);
```

Purpose

Determines whether a file name, **str**, matches a pattern, **pat**.

Parameters

Name	Type	Description
pat	`uChar *`	Pattern (string) to which filename is to be compared. The following characters have special meanings in the pattern. `\` is literal, not treated as having a special meaning. A single backslash at the end of **pat** is the same as two backslashes. `?` matches any one character. `*` matches zero or more characters.
str	`uChar *`	File name (string) to compare to pattern.
pLen	`int32`	Number of characters in **pat**.
sLen	`int32`	Number of characters in **str**.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	File name fits the pattern.
`FALSE`	File name does not match the pattern.

FStringToPath

```
MgErr FStringToPath(text, p);
```

Purpose

Creates a path from an `LStr`. The `LStr` contains the platform-specific syntax for a path.

Parameters

Name	Type	Description
text	`LstrHandle`	String that contains the path in platform-specific syntax.
p	`Path *`	Address at which `FstringToPath` stores the resulting path. If nonzero, the function assumes it is a valid path, resizes the path, and fills in its value. If `NULL`, the function creates a new path, fills in its value, and stores the path at the address referred to by **p**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mFullErr`	Insufficient memory.

FTextToPath

```
MgErr FTextToPath(text, tlen, *p);
```

Purpose

Creates a path from a string at the address **text** that represents a path in the platform-specific syntax for a path.

Parameters

Name	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 nonzero, the function assumes it is a valid path, resizes the path, and fills in its value. If `NULL`, the function creates a new path, fills in its value, and stores the path at the address referred to by **p**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mFullErr`	Insufficient memory.

FUnFlattenPath

```
int32 FUnFlattenPath(fp, pPtr);
```

Purpose

Converts a flattened path, created using `FFlattenPath`, into a path.

Parameters

Name	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. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

Return Value

Number of bytes the function interpreted as a path.

FVolName

`MgErr FVolName(path, vol);`

Purpose

Creates a path for the volume of an absolute path by removing all but the first component name from **path**. You can pass the same path variable for the volume path that you use for **path**. Therefore, you can call this function in the following two ways:

- `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 */`

Parameters

Name	Type	Description
path	`Path`	Path whose volume path you want to determine.
vol	`Path`	Parameter in which `FVolName` stores the volume path.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mgArgErr`	A bad argument was passed to the function. Verify the path.

GetALong

Macro

```
int32 GetALong(p);
```

Purpose

Retrieves an `int32` from a `void` pointer. In SPARCstation, this function can retrieve an `int32` at any address, even if the `int32` is not long word aligned.

Parameters

Name	Type	Description
p	`void *`	Address from which you want to read an `int32`.

Return Value

`int32` stored at the specified address.

HexChar

```
int32 HexChar(n);
```

Purpose

Returns the ASCII character in hex that represents the specified value **n**, 0 ≤ **n** ≤ 15.

Parameters

Name	Type	Description
n	`int32`	Decimal value between `0` and `15`.

Return Value

The corresponding ASCII hex character. If **n** is out of range, the function returns the ASCII character corresponding to **n** modulo 16.

Hi16

Macro

```
int16 Hi16(x);
```

Purpose

Returns the high order `int16` of an `int32`.

Parameters

Name	Type	Description
x	`int32`	`int32` for which you want to determine the high `int16`.

HiByte

Macro

```
int8 HiByte(x);
```

Purpose

Returns the high order `int8` of an `int16`.

Parameters

Name	Type	Description
x	`int16`	`int16` for which you want to determine the high `int8`.

HiNibble

Macro

```
uInt8 HiNibble(x);
```

Purpose

Returns the value stored in the high four bits of an `uInt8`.

Parameters

Name	Type	Description
x	`uInt8`	`uInt8` whose high four bits you want to extract.

IsAlpha

```
Bool32 IsAlpha(c);
```

Purpose

Returns `TRUE` if the character **c** is a lowercase or uppercase letter, that is, in the set `a` to `z` or `A` to `Z`. In SPARCstation, this function also returns `TRUE` for international characters, such as `à`, `á`, `Ä`, and so on.

Parameters

Name	Type	Description
c	`uChar`	Character you want to analyze.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	The character is alphabetic.
`FALSE`	Otherwise.

IsDigit

```
Bool32 IsDigit(c);
```

Purpose

Returns `TRUE` if the character **c** is between `0` and `9`.

Parameters

Name	Type	Description
c	`uChar`	Character you want to analyze.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	Character is a numerical digit.
`FALSE`	Otherwise.

IsLower

```
Bool32 IsLower(c);
```

Purpose

Returns `TRUE` if the character **c** is a lowercase letter, that is, in the set `a` to `z`. In SPARCstation, this function also returns `TRUE` for lowercase international characters, such as `ó`, `ö`, and so on.

Parameters

Name	Type	Description
c	`uChar`	Character you want to analyze.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	Character is a lowercase letter.
`FALSE`	Otherwise.

IsUpper

`Bool32 IsUpper(c);`

Purpose

Returns `TRUE` if the character **c** is between an uppercase letter, that is, in the set `A` to `Z`. In SPARCstation, this function also returns `TRUE` for uppercase international characters, such as `Ó`, `Ä`, and so on.

Parameters

Name	Type	Description
c	`uChar`	Character you want to analyze.

Return Value

`Bool32`, which can contain the following values:

`TRUE`	Character is an uppercase letter.
`FALSE`	Otherwise.

Lo16

Macro

```
int16 Lo16(x);
```

Purpose

Returns the low order `int16` of an `int32`.

Parameters

Name	Type	Description
x	`int32`	`int32` for which you want to determine the low `int16`.

LoByte

Macro

```
int8 LoByte(x);
```

Purpose

Returns the low order `int8` of an `int16`.

Parameters

Name	Type	Description
x	`int16`	`int16` for which you want to determine the low `int8`.

Long

Macro

```
int32 Long(hi, lo);
```

Purpose

Creates an `int32` from two `int16` parameters.

Parameters

Name	Type	Description
hi	`int16`	High `int16` for the resulting `int32`.
lo	`int16`	Low `int16` for the resulting `int32`.

Return Value

The resulting `int32`.

LoNibble

Macro

```
uInt8 LoNibble(x);
```

Purpose

Returns the value stored in the low four bits of an `uInt8`.

Parameters

Name	Type	Description
x	`uInt8`	`uInt8` whose low four bits you want to extract.

LStrBuf

Macro

```
uChar *LStrBuf(s);
```

Purpose

Returns the address of the string data of a long Pascal string, that is, the address of `s->str`.

Parameters

Name	Type	Description
s	`LStrPtr`	Pointer to a long Pascal string.

Return Value

The address of the string data of the long Pascal string.

LStrCmp

```
LStrPtr LStrCmp(l1p, l2p);
```

Purpose

Lexically compares two long Pascal strings to determine whether one is less than, equal to, or greater than the other. This comparison is case sensitive.

Parameters

Name	Type	Description
l1p	`LStrPtr`	Pointer to a long Pascal string.
l2p	`LStrPtr`	Pointer to a long Pascal string.

Return Value

`<0`, **`0`**, or **`>0`** if **l1p** is less than, equal to, or greater than **l2p**, respectively. Returns **`<0`** if **l1p** is an initial substring of **l2p**.

LStrLen

Macro

```
int32 LStrLen(s);
```

Purpose

Returns the length of a long Pascal string, that is, `s->cnt`.

Parameters

Name	Type	Description
s	`LStrPtr`	Pointer to a long Pascal string.

Return Value

The number of characters in the long Pascal string.

LToPStr

```
int32 LToPStr(lstrp, pstr);
```

Purpose

Converts a long Pascal string to a Pascal string. If the long Pascal string is more than 255 characters, this function converts only the first 255 characters. The function works even if the pointers **lstrp** and **pstr** refer to the same memory location. The function assumes **pstr** is large enough to contain **lstrp**.

Parameters

Name	Type	Description
lstrp	`LStrPtr`	Pointer to a long Pascal string.
pstr	`PStr`	Pointer to a Pascal string.

Return Value

The length of the string, truncated to a maximum of 255 characters.

Max

```
int32 Max(n,m);
```

Purpose

Returns the maximum of two `int32` parameters.

Parameters

Name	Type	Description
n, m	`int32`	`int32` parameters whose maximum value you want to determine.

MilliSecs

```
uint32 MilliSecs();
```

Return Value

The time in milliseconds since an undefined system time. The actual resolution of this timer is system dependent.

Min

```
int32 Min(n,m);
```

Purpose

Returns the minimum of two `int32` parameters.

Parameters

Name	Type	Description
n, m	`int32`	`int32` parameters whose minimum value you want to determine.

MoveBlock

```
void MoveBlock(ps, pd, size);
```

Purpose

Moves **size** bytes from one address to another. The source and destination memory blocks can overlap.

Parameters

Name	Type	Description
ps	`UPtr`	Pointer to source.
pd	`UPtr`	Pointer to destination.
size	`int32`	Number of bytes you want to move.

NumericArrayResize

```
MgErr NumericArrayResize (int32 typeCode, int32 numDims, Uhandle *dataHP,
                           int32 totalNewSize)
```

Purpose

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**.

Parameters

Name	Type	Description
typeCode	`int32`	Data type for the array you want to resize.
numDims	`int32`	Number of dimensions in the data structure to which the handle refers.
***dataHP**	`UHandle`	Pointer to the handle you want to resize.
totalNewSize	`int32`	New number of elements to which the handle should refer.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`MFullErr`	Not enough memory to perform the operation.
`mZoneErr`	Handle or pointer not in specified zone.

Possible Values for this Type Code Input

Data Type	Type Code (numbers in hexadecimal)
8-bit integer	01 or iB
16-bit integer	02 or iW
32-bit integer	03 or iL
8-bit unsigned integer	05 or uB
16-bit unsigned integer	06 or uW
32-bit unsigned integer	07 or uL
Single-precision, floating-point number	09 or fs

Data Type	Type Code (numbers in hexadecimal)
Double-precision, floating-point number	0A or fD
Extended-precision, floating-point number	0B or fX
Complex single-precision, floating-point number	0C or cS
Complex double-precision, floating-point number	0D or cD
Complex extended-precision, floating-point number	0E or cX

Occur

```
MgErr Occur(Ocurrence o);
```

Purpose

Triggers the specified occurrence. All block diagrams that are waiting for this occurrence stop waiting.

Parameters

Name	Type	Description
o	`Occurrence`	Occurrence refnum you want to trigger.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mgArgErr`	Not a valid user event.

Offset

Macro

```
int16 Offset(type, field);
```

Purpose

Returns the offset of the specified field within the structure called **type**.

Parameters

Name	Type	Description
type	—	Structure that contains **field**.
field	—	Field whose offset you want to determine.

Return Value

An offset as an `int16`.

Pin

```
int32 Pin(i, low, high);
```

Purpose

Returns **i** coerced to fall within the range from **low** to **high** inclusive.

Parameters

Name	Type	Description
i	`int32`	Value you want to coerce to the specified range.
low	`int32`	Low value of the range to which you want to coerce **i**.
high	`int32`	High value of the range to which you want to coerce **i**.

Return Value

i coerced to the specified range.

PostLVUserEvent

```
MgErr PostLVUserEvent(LVUserEventRef ref, void *data);
```

Purpose

Posts the given user event. The event and associated data are queued for all event structures registered for the event.

Parameters

Name	Type	Description
ref	`LVUserEventRef`	Event refnum for the event for which you want to post data.
data	`void*`	Address of the data to post. The data must match the type used to create the user event.

Return Value

`MgErr`, which can contain the following error:

`noErr`	No error.
`mgArgErr`	Not a valid user event.

PPStrCaseCmp

```
int32 PPStrCaseCmp(s1p, s2p);
```

Purpose

Lexically compares two Pascal strings and determines whether one is less than, equal to, or greater than the other. This comparison ignores differences in case. This function is similar to `PStrCaseCmp`, except you pass the function handles to the string data instead of pointers.

Parameters

Name	Type	Description
s1p	`PStr *`	Pointer to a Pascal string.
s2p	`PStr *`	Pointer to a Pascal string.

Return Value

`<0`, **`0`**, or **`>0`** if **s1p** is less than, equal to, or greater than **s2p**, respectively. Returns **`<0`** if **s1p** is an initial substring of **s2p**.

PPStrCmp

```
int32 PPStrCmp(s1p, s2p);
```

Purpose

Lexically compares two Pascal strings and determines whether one is less than, equal to, or greater than the other. This comparison is case sensitive. This function is similar to `PStrCmp`, except you pass the function handles to the string data instead of pointers.

Parameters

Name	Type	Description
s1p	`PStr *`	Pointer to a Pascal string.
s2p	`PStr *`	Pointer to a Pascal string.

Return Value

`<0`, **`0`**, or **`>0`** if **s1p** is less than, equal to, or greater than **s2p**, respectively. Returns **`<0`** if **s1p** is an initial substring of **s2p**.

Printf

SPrintf, SPrintfp, PPrintf, PPrintfp, FPrintf, LStrPrintf

```
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,...);
```

Purpose

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. 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 and SPrintfp

`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. `SPrintf` and `SPrintfp` return the number of characters required to contain the resulting data, not including the terminating `NULL` character.

PPrintf and PPrintfp

`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

`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

`LStrPrintf` prints to a LabVIEW string specified by **destLsh**. Because the string is a handle that might be resized, `LStrPrintf` can return memory errors just as `DSSetHandleSize` does.

Special Chracters

These functions accept the following special characters:

`\b`	Backspace
`\f`	Form feed
`\n`	New line, which inserts the system-dependent end-of-line char(s); for example, CR on Mac OS, NL on UNIX, CRNL on DOS)
`\r`	Carriage return
`\s`	Space
`\t`	Tab
`%%`	Percentage character (to print %)

Formats

These functions accept the following formats:

`%[-]`	Left-justifies what is printed; if not specified, the data is right-justified.
`[field size]`	Indicates the minimum width of the field to print into. If not specified, the default is `0`. If less than the specified number of characters are 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]`	Indicates 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 16-bit integer. The flag l causes this conversion to convert the data so the function assumes the data is a 32-bit integer value.
`[conversion]`	You can precede any of the numeric conversion characters (`x`, `o`, `d`, `u`, `b`, `e`, `f`) 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.

Conversion Specifiers

These functions use the conversion specifiers listed and described in Table 6-1.

Table 6-1. Conversion Specifiers and Descriptions

Conversion Specifier	Description
`b`	Binary integer
`c`	Character
`d`	Decimal integer
`e`, `E`	Single-precision, floating-point number in scientific notation
`f`	Single-precision, floating-point number
`F`	Double-precision, floating-point number
`g`, `G`	Double-precision floating-point number in scientific notation
`H`	String handle (`LStrHandle`)
`o`	Octal integer
`p`	Pascal string (`PStr`)
`P`	Long Pascal string (`LStrPtr`)
`q`	Point (passed by value) as `%d,%d` representing horizontal, vertical coordinates
`Q`	Point (passed by value) as `hv(%d,%d)` representing horizontal, vertical coordinates
`r`	Rectangle (passed by reference) as `%d,%d,%d,%d` representing top, left, bottom, right coordinates
`R`	Rectangle (passed by reference) as `tlbr(%d,%d,%d,%d)` representing top, left, bottom, right coordinates
`s`	C String (null-terminated)
`u`	Unsigned decimal integer
`x`	Hex integer
`z`	Path

PStrBuf

Macro

```
uChar *PStrBuf(s);
```

Purpose

Returns the address of the string data of a Pascal string, that is, the address following the length byte.

Parameters

Name	Type	Description
s	`PStr`	Pointer to a Pascal string.

PStrCaseCmp

```
int32 PStrCaseCmp(s1, s2);
```

Purpose

Lexically compares two Pascal strings to determine whether one is less than, equal to, or greater than the other. This comparison ignores differences in case.

Parameters

Name	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

Return Value

`<0`, **`0`**, or **`>0`** if **s1** is less than, equal to, or greater than **s2**, respectively. Returns **`<0`** if **s1** is an initial substring of **s2**.

PStrCat

```
int32 PStrCat(s1, s2);
```

Purpose

Concatenates a Pascal string, **s2**, to the end of another Pascal string, **s1**, and returns the result in **s1**. This function assumes **s1** is large enough to contain the resulting string. If the resulting string is larger than 255 characters, the function limits the resulting string to 255 characters.

Parameters

Name	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

Return Value

The length of the resulting string.

PStrCmp

```
int32 PStrCmp(s1, s2);
```

Purpose

Lexically compares two Pascal strings to determine whether one is less than, equal to, or greater than the other. This comparison is case sensitive.

Parameters

Name	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

Return Value

`<0`, **`0`**, or **`>0`** if **s1** is less than, equal to, or greater than **s2**, respectively. Returns **`<0`** if **s1** is an initial substring of **s2**.

PStrCpy

```
PStr PStrCpy(dst, src);
```

Purpose

Copies the Pascal string **src** to the Pascal string **dst**. This function assumes **dst** is large enough to contain **src**.

Parameters

Name	Type	Description
dst	`PStr`	Pointer to a Pascal string.
src	`PStr`	Pointer to a Pascal string.

Return Value

A copy of the destination Pascal string pointer.

PStrLen

Macro

```
uInt8 PStrLen(s);
```

Purpose

Returns the length of a Pascal string, that is, the value at the first byte at the specified address.

Parameters

Name	Type	Description
s	`PStr`	Pointer to a Pascal string.

PStrNCpy

```
PStr PStrNCpy(dst, src, n);
```

Purpose

Copies the Pascal string **src** to the Pascal string **dst**. If the source string is greater than **n**, this function copies only **n** bytes. The function assumes **dst** is large enough to contain **src**.

Parameters

Name	Type	Description
dst	`PStr`	Pointer to a Pascal string.
src	`PStr`	Pointer to a Pascal string.
n	`int32`	Maximum number of bytes you want to copy, including the length byte.

Return Value

A copy of the destination Pascal string pointer.

PToCStr

```
int32 PToCStr(pstr, cstr);
```

Purpose

Converts a Pascal string to a C string. This function works even if the pointers **pstr** and **cstr** refer to the same memory location. The function assumes **cstr** is large enough to contain **pstr**.

Parameters

Name	Type	Description
pstr	`PStr`	Pointer to a Pascal string.
cstr	`CStr`	Pointer to a C string.

Return Value

The length of the string.

PToLStr

```
int32 PToLStr(pstr, lstrp);
```

Purpose

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 **lstrp** is large enough to contain **pstr**.

Parameters

Name	Type	Description
pstr	`PStr`	Pointer to a Pascal string.
lstrp	`LStrPtr`	Pointer to a long Pascal string.

Return Value

The length of the string.

QSort

```
void QSort(arrayp, n, elmtSize, compareProcP());
```

Purpose

Sorts an array of an arbitrary data type using the QuickSort algorithm. In addition to passing the array 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);
```

Parameters

Name	Type	Description
arrayp	`UPtr`	Pointer to an array of data.
n	`int32`	Number of elements in the array you want to sort.
elmtSize	`int32`	Size in bytes of an array element.
compareProcP	`CompareProcPtr`	Comparison routine you want `QSort` to use to compare array elements. `QSort` passes this routine the addresses of two elements that it needs to compare.

RandomGen

```
void RandomGen(xp);
```

Purpose

Generates a random number between `0` and `1` and stores it at **xp**.

Parameters

Name	Type	Description
xp	`float64 *`	Location to store the resulting double-precision floating-point random number. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

SecsToDate

```
void SecsToDate(secs, dateRecordP);
```

Purpose

Converts the seconds since January 1, 1904, 12:00 a.m., UT into a data structure containing numerical information about the date, including the year (`1904` through `2040`), the month (`1` through `12`), the day of the year (`1` through `366`), the day of the month (`1` through `31`), the day of the week (`1` through `7`), the hour (`0` through `23`), the minute (`0` through `59`), and the second (`0` through `59`), and a value indicating whether the time specified uses daylight savings time.

Parameters

Name	Type	Description
secs	`uInt32`	Seconds since January 1, 1904, 12:00 a.m., UT.
dateRecordP	`DateRec *`	Pointer to a `DateRec` structure. `SecsToDate` stores the converted date in the fields of the date structure referred to by **dateRecordP**. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.

SetALong

Macro

```
void SetALong(p,x);
```

Purpose

Stores an `int32` at the address specified by a void pointer. In SPARCstation, this function can retrieve an `int32` at any address, even if it is not long word aligned.

Parameters

Name	Type	Description
p	`void *`	Address at which you want to store an `int32`. Refer to the *Pointers as Parameters* section of Chapter 3, *CINs*, for more information about using this parameter.
x	`int32`	Value you want to store at the specified address.

SetCINArraySize

```
MgErr SetCINArraySize (Uhandle dataH, int32 paramNum, int32 newNumElmts)
```

Purpose

Resizes a data handle based on the data structure of an argument that you pass to the CIN. This function does not set the array dimension field.

Parameters

Name	Type	Description
dataH	`UHandle`	Handle you want to resize.
paramNum	`int32`	Number for this parameter in the argument list to the CIN.
newNumElmts	`int32`	New number of elements to which the handle refers.

Return Value

`MgErr`, which can contain the following errors:

`noErr`	No error.
`MFullErr`	Not enough memory to perform the operation.
`mZoneErr`	Handle or pointer not in specified zone.

StrCat

```
int32 StrCat(s1, s2);
```

Purpose

Concatenates a C string, **s2**, to the end of another C string, **s1**, returning the result in **s1**. This function assumes **s1** is large enough to contain the resulting string.

Parameters

Name	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.

Return Value

The length of the resulting string.

StrCmp

```
int32 StrCmp(s1, s2);
```

Purpose

Lexically compares two strings to determine whether one is less than, equal to, or greater than the other.

Parameters

Name	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.

Return Value

`<0`, **`0`**, or **`>0`** if **s1** is less than, equal to, or greater than **s2**, respectively. Returns **`<0`** if **s1** is an initial substring of **s2**.

StrCpy

```
CStr StrCpy(dst, src);
```

Purpose

Copies the C string **src** to the C string **dst**. This function assumes **dst** is large enough to contain **src**.

Parameters

Name	Type	Description
dst	`CStr`	Pointer to a C string.
src	`CStr`	Pointer to a C string.

Return Value

A copy of the destination C string pointer.

StrLen

```
int32 StrLen(s);
```

Purpose

Returns the length of a C string.

Parameters

Name	Type	Description
s	`CStr`	Pointer to a C string.

Return Value

The number of characters in the C string, not including the `NULL` terminating character.

StrNCaseCmp

`int32 StrNCaseCmp(s1, s2, n);`

Purpose

Lexically compares two strings to determine whether one is less than, equal to, or greater than the other, limiting the comparison to **n** characters. This comparison ignores differences in case.

Parameters

Name	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.
n	`uInt32`	Maximum number of characters you want to compare.

Return Value

`<0`, **`0`**, or **`>0`** if **s1** is less than, equal to, or greater than **s2**, respectively. Returns **`<0`** if **s1** is an initial substring of **s2**.

StrNCmp

`int32 StrNCmp(s1, s2, n);`

Purpose

Lexically compares two strings to determine whether one is less than, equal to, or greater than the other, limiting the comparison to **n** characters.

Parameters

Name	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.
n	`uInt32`	Maximum number of characters you want to compare.

Return Value

`<0`, **`0`**, or **`>0`** if **s1** is less than, equal to, or greater than **s2**, respectively. Returns **`<0`** if **s1** is an initial substring of **s2**.

StrNCpy

```
CStr StrNCpy(dst, src, n);
```

Purpose

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**, only **n** characters are copied. This function assumes **dst** is large enough to contain **src**.

Parameters

Name	Type	Description
dst	`CStr`	Pointer to a C string.
src	`CStr`	Pointer to a C string.
n	`int32`	Maximum number of characters you want to copy.

Return Value

A copy of the destination C string pointer.

SwapBlock

```
void SwapBlock(ps, pd, size);
```

Purpose

Swaps **size** bytes between the section of memory referred to by **ps** and **pd**. The source and destination memory blocks should not overlap.

Parameters

Name	Type	Description
ps	`UPtr`	Pointer to source.
pd	`UPtr`	Pointer to destination.
size	`int32`	Number of bytes you want to move.

TimeCString

```
CStr TimeCString(secs, fmt);
```

Purpose

Returns a pointer to a string representing the time of day corresponding to *t* seconds after January 1, 1904, 12:00 a.m., UT. In SPARCstation, this function accounts for international conventions for representing dates.

Note This function was formerly called `TimeString`.

Parameters

Name	Type	Description
secs	`uInt32`	Seconds since January 1, 1904, 12:00 a.m., UT.
fmt	`int32`	Indicates the format of the returned time string, using the following values: • `0`—*`hh`*`:`*`mm`* format, where *`hh`* is the hour (`0` through `23`, with `0` as midnight), and the *`mm`* is the minute (`0` through `59`). • `1`—*`hh`*`:`*`mm`*`:`*`ss`* format, where *`hh`* is the hour, *`mm`* is the minute (`0` through `59`), and *`ss`* is the second (`0` through `59`).

Return Value

The time as a C string.

TimeInSecs

```
uint32 TimeInSecs();
```

Return Value

The current date and time in seconds relative to January 1, 1904, 12:00 a.m., UT.

ToLower

```
uChar ToLower(c);
```

Purpose

Returns the lowercase value of **c** if **c** is an uppercase alphabetic character. Otherwise, this function returns **c** unmodified. In SPARCstation, this function also works for international characters (`Ä` to `ä`, and so on).

Parameters

Name	Type	Description
c	`int32`	Character you want to analyze.

Return Value

The lowercase value of **c**.

ToUpper

```
uChar ToUpper(c);
```

Purpose

Returns the uppercase value of **c** if **c** is a lowercase alphabetic character. Otherwise, this function returns **c** unmodified. In SPARCstation, this function also works for international characters (`ä` to `Ä`, and so on).

Parameters

Name	Type	Description
c	`int32`	Character you want to analyze.

Return Value

The uppercase value of **c**.

Unused

Macro

```
void Unused(x)
```

Purpose

Indicates that a function parameter or local variable is not used by that function. This is useful for suppressing compiler warnings for many compilers. This macro does not use a semicolon.

Parameters

Name	Type	Description
x	—	Unused parameter or local variable.

Word
Macro

```
int16 Word(hi, lo);
```

Purpose

Creates an `int16` from two `int8` parameters.

Parameters

Name	Type	Description
hi	`int8`	High `int8` for the resulting `int16`.
lo	`int8`	Low `int8` for the resulting `int16`.

Return Value

The resulting `int16`.

A

Technical Support and Professional Services

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

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

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

Glossary

A

ANSI	American National Standards Institute.
application zone	*See* AZ (application zone).
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.
AZ (application zone)	Memory allocation section that holds all data in a VI except execution data.

B

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.
CIN source code	Original, uncompiled text code. *See also* object code; Code Interface Node.
Code Interface Node	Special block diagram node through which you can link conventional, text-based code to a VI.
code resource	Resource containing executable machine code. You link code resources to LabVIEW through a CIN.
concatenated Pascal string (CPStr)	A list of Pascal-type strings concatenated into a single block of memory.
CPStr	*See* concatenated Pascal string (CPStr).

D

data type descriptor	Code that identifies data types, used in data storage and representation.
diagram window	VI window containing the VI's block diagram code.
dimension	Size and structure attribute of an array.

E

executable	A stand-alone piece of code that will run, or execute.

I

icon pane	Region in the upper right-hand corner of the front panel and block diagram windows that displays the VI icon.
IDE	Integrated development environment for developing computer applications, for example, Visual Basic, Visual C++, and LabVIEW.
inplace	When the input and output data of an operation use the same memory space.

L

LabVIEW string	The string data type (LStr) that LabVIEW block diagrams use.

M

MB	Megabytes of memory.
MPW	Macintosh Programmer's Workshop.
MSB	Most significant bit.

O

object code	Compiled version of source code. Object code is not standalone because you must load it into LabVIEW to run it.

P

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.
portable	Able to compile on any platform that supports LabVIEW.
private data structures	Data structures whose exact format is not described; usually subject to change.

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.
relocatable	Able to be moved by the memory manager to a new memory location.

S

sink terminal	Terminal that absorbs data. Also called a destination terminal.
shortcut menu	Menu that you access by right-clicking an object. Menu items pertain to that object specifically.
source code	Original, uncompiled text code.
source terminal	Terminal that emits data.

T

type descriptor	*See* data type descriptor.

U

UT	Universal Time.