NATIONAL INSTRUMENTS™
LabVIEW™

# Using External Code in LabVIEW

NATIONAL INSTRUMENTS™

July 2000 Edition
Part Number 370109A-01

**Worldwide Technical Support and Product Information**

www.ni.com

**National Instruments Corporate Headquarters**

11500 North Mopac Expressway Austin, Texas 78759-3504 USA Tel: 512 794 0100

**Worldwide Offices**

Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 284 5011,
Canada (Calgary) 403 274 9391, Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521,
China 0755 3904939, Denmark 45 76 26 00, Finland 09 725 725 11, France 01 48 14 24 24,
Germany 089 741 31 30, Greece 30 1 42 96 427, Hong Kong 2645 3186, India 91805275406,
Israel 03 6120092, Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico (D.F.) 5 280 7625,
Mexico (Monterrey) 8 357 7695, Netherlands 0348 433466, New Zealand 09 914 0488, Norway 32 27 73 00,
Poland 0 22 528 94 06, Portugal 351 1 726 9011, Singapore 2265886, Spain 91 640 0085,
Sweden 08 587 895 00, Switzerland 056 200 51 51, Taiwan 02 2528 7227, United Kingdom 01635 523545

For further support information, see the *Technical Support Resources* appendix. To comment on the documentation, send e-mail to techpubs@ni.com

# Important Information

## Warranty

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

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

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

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

## Copyright



## Trademarks

CVI™, LabVIEW™, National Instruments™, and ni.com™ are trademarks of National Instruments Corporation.

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

## WARNING REGARDING USE OF NATIONAL INSTRUMENTS PRODUCTS

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

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

# Contents

## About This Manual



## Chapter 1
## Introduction



## Chapter 2
## Shared Libraries (DLLs)

# Chapter 3
# CINs



# Chapter 4
# Programming Issues for CINs

# Chapter 5
# Advanced Applications



# Chapter 6
# Function Descriptions

# About This Manual

This manual describes the LabVIEW programming objects that you use to call compiled code from text-based programming languages: the Call Library Function and the Code Interface Node. 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 on 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 document contains information that you might find helpful as you read this manual:

- *LabVIEW User Manual*

Sun users might also find the following document helpful:

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

Linux users might also find the following document helpful:

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

Windows users might also 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 several methods in LabVIEW to call code written in other languages.

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

Also, to convert an instrument driver written in LabWindows/CVI you can select **Tools»Instrumentation»Import CVI Instrument Driver** and invoke the **LabWindows/CVI Function Panel Converter**. Refer to *LabVIEW Help* for more information about this converter.

## Calling Code in Various Platforms

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

**(Windows and UNIX)** Use the System Exec VI. Use the simple System Exec VI on the **Functions»Communication** palette 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.

**(Macintosh)** Use the Apple Event VIs. Apple Events are a Macintosh-specific protocol through which applications communicate with each other. You can use them to send commands between applications or to launch other applications. 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 Apple Event VIs to launch and control other applications.

# Characteristics of the Two Calling Approaches

**Note** In most cases, Call Library Function is easier to use than a Code Interface Node.

The LabVIEW Call Library Function and the Code Interface Node (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 steps:

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 node 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 Macintosh, these objects 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 executes.

## Details of Call Library Function

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

Call Library Function 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 the Chapter 2, *Shared Libraries (DLLs)*, for more information about Call Library Function.

## 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.

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, there are restrictions on 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.

# 2

# Shared Libraries (DLLs)

This chapter describes how to call shared libraries—called DLLs on the Windows platform—from LabVIEW. Examples and troubleshooting information appear later in the chapter to help you build and use DLLs and configure LabVIEW's Call Library Function successfully. The general methods described here for DLLs also apply to other types of shared libraries.

## Calling Shared Libraries

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

The diagram on the left shows the Call Library Function object on the block diagram. You access this function on the **Functions»Advanced** palette.

Right-click the icon and select **Configure** in the shortcut menu to access the **Call Library Function** dialog box where you specify the library, function, parameters, return value for the object, and calling conventions in Windows. When you click **OK** in the dialog box, LabVIEW updates the icon according to your settings, displaying the correct number of terminals and setting the terminals to the correct data types. The following figure shows the **Call Library Function** dialog box.

**Note** The shortcut menu for the Call Library Function object also contains the **Create .c File** item, which creates a `.c` prototype file that contains C declarations for the parameters that you are passing.

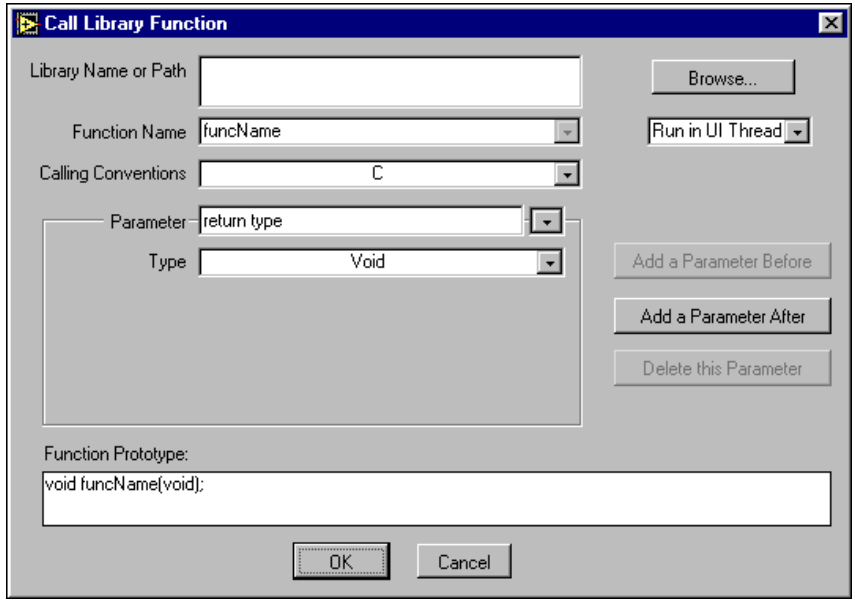



**Figure 2-1.** Call Library Function Dialog Box

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.

The return value for the function returns to the right terminal of the top pair of terminals of the object. If there is no return value, this pair of terminals is unused. Each additional pair of terminals corresponds to a parameter in the functions parameter list. To pass a value to the function, wire to the left terminal of a terminal pair. To read the value of a parameter after the function call, wire from the right terminal of a terminal pair.

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 object to be reentrant, make sure that multiple threads can call the function(s) simultaneously. The following list shows the basic characteristics of thread safe code in a shared library.

- The code is thread safe when it stores no global data (for example, no global variables, no files on disk, and so on); does not access any

hardware (in other words, does not contain register-level programming); and makes no 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 in *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.

## Calling Conventions (Windows)

Use the **Calling Conventions** pull-down menu in the **Call Library Function** dialog box to select the calling conventions for the function. The default calling convention is C. You can also use the standard Windows calling convention, `__stdcall`. Refer to the documentation for the DLL you want to call for the appropriate calling conventions.

## Parameters

Initially, Call Library Function has no parameters and has a return value of **Void**. To add parameters to the function, click the **Add a Parameter Before** or **After** buttons. To remove a parameter, click the **Delete this Parameter** button.

Use the **Parameter** pull-down menu to select different parameters or the return value. 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.

Use the **Type** pull-down menu to indicate the type of each parameter. The return type can be **Void**, meaning the function does not return a value, **Numeric**, or **String**.

For parameters, you can select **Numeric**, **Array**, **String**, **Waveform**, **ActiveX**, or **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 data type and about how to pass the data to the library function. Call Library Function has a number of different items for data types, because of the variety of data types required

by different libraries. Refer to the documentation for the library you call to determine which data types to use.

- **Void**—Accepted only for the return value. This value is not available for parameters. If your function does not return any values, use **Void** for the return value.
- **Numerics**—For numeric data types, you must indicate the exact numeric type using the **Data Type** pull-down menu. Valid types include the following:
  - Signed and unsigned 8-bit, 16-bit, and 32-bit integers
  - Four-byte, single-precision numbers
  - Eight-byte, double-precision numbers

  You cannot use extended-precision numbers and complex numbers. Standard libraries generally do not use them.

  You also must use the **Pass** pulldown menu to indicate whether you want to pass the value or a pointer to the value.
- **Arrays**—Indicate the data type of arrays using the same items as for numeric data types, the number of dimensions, and the format to use in passing the array. Use the **Array Format** pull-down menu to make one of the following choices:
  - **Array Data Pointer**—Passes a pointer to the array data.
  - **Array Handle**—Passes a pointer to a pointer 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 CIN manager functions, such as `NumericArrayResize`.

**Strings**—Indicate the string format for strings. Valid values for **String Format** include **C String Pointer**, **Pascal String Pointer**, **String Handle**, or **String Handle Pointer**.

Select a string format that the library function expects. Most standard libraries expect either a C string (string followed by a null character) or a Pascal string (string preceded by a length byte). If the library function you are calling is written for LabVIEW, you might want to use the **String Handle** format, which is a pointer to a pointer to four bytes for length information, followed by string data.

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

**Waveform**—For waveform data types, you indicate the dimension, and you use the **Data Type** pull-down menu to indicate the exact numeric type.

**ActiveX**—For ActiveX objects, you select one of the following items in 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**—Pass arbitrary LabVIEW data types to DLLs in the same way they are passed to a CIN, as follows:

- Scalars are passed by reference. A pointer to the scalar is passed to the library.
- Arrays and strings are passed as a handle (pointer to a pointer to the data).
- 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.
- Cluster within arrays are in line.
- Strings and arrays within clusters are referenced by a handle.

## 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 Call Library Function to pass an arbitrary cluster or array of non-numeric 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, and parameters to build the data structures the library function expects, then calls 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)

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

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

In the *Example 1: Call a Shared Library that You Built* section, you will 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 will be valid.

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

1. Create a VI called Array Average in LabVIEW, and access the block diagram. Select **Functions»Advanced»Call Library Function** and place this object on the block diagram.
2. Right-click the Call Library Function icon and select **Configure** in the shortcut menu to invoke the **Call Library Function** dialog box. Leave the **Library Name or Path** control empty.
3. Enter the following general specifications.
   a. Type `avg_num` in the **Function Name** field.
   b. Select **C** in the **Calling Conventions** control.
4. Define the return value:
   a. In the **Parameter** control change the default name, `return type`, to a more descriptive name, `error`.
   b. In the **Type** control select **Numeric**.
   c. In the **Data Type** control select **Signed 32-bit Integer**.

5. Define the a parameter:
   a. Click the **Add Parameter After** button.
   b. In the **Parameter** control replace the default name, `arg1`, with the precise name, `a`.
   c. In the **Type** control select **Array**.
   d. In the **Data Type** control select **4-byte Single**.
   e. In the **Array Format** control select **Array Data Pointer**.

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

6. Define `size`:
   a. Click the **Add Parameter After** button.
   b. In the **Parameter** control replace the default name, `arg2`, with the precise name, `size`.
   c. In the **Type** control select **Numeric**.
   d. In the **Data Type** control select **Signed 32-bit Integer**.
   e. In the **Pass** control select **Value**.
7. Define `avg`:
   a. Click the **Add Parameter After** button.
   b. In the **Parameter** control replace the default name, `arg3`, with the precise name, `avg`.
   c. In the **Type** control select **Numeric** type.
   d. In the **Data Type** control select **4-byte Single**.
   e. In the **Pass** control select **Pointer to Value**.
8. Check that the Function Prototype indicator 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** indicator is technically correct. However, the `.c` file that LabVIEW generates in the next section will be more precise because the first parameter will appear as `float a[]`.

9. Click **OK** to save your settings and close the dialog box.
10. Observe how the Call Library Function icon updates to reflect your settings.

11. Right-click the Call Library Function icon and select **Create .c file** in the shortcut menu. 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 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 LabVIEW library files in your compiled project: `labview.lib` for Visual C++, `labview.sym.lib` for Symantec, and `labview.export.stub` for Metrowerks CodeWarrior. These 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:

- 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. Therefore, this section describes three compiler/platform combinations that you can use to build shared libraries to use in LabVIEW: Microsoft Visual C/C++ on Windows, Gnu C/C++ on UNIX, and Metrowerks CodeWarrior on Macintosh.

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

Follow the steps in this section to build a project and 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. By declaring the dllexport keyword, you eliminate the need for a module definition file, which the *Module Definition Files* section describes. `_declspec (dllexport)` is a Microsoft-specific extension to the C or C++ language.

1. 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

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

2. Select **File»New** and select **Win32 Dynamic Link Library** (DLL) in the listbox of the Projects tab. Click **OK** to continue.

**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.

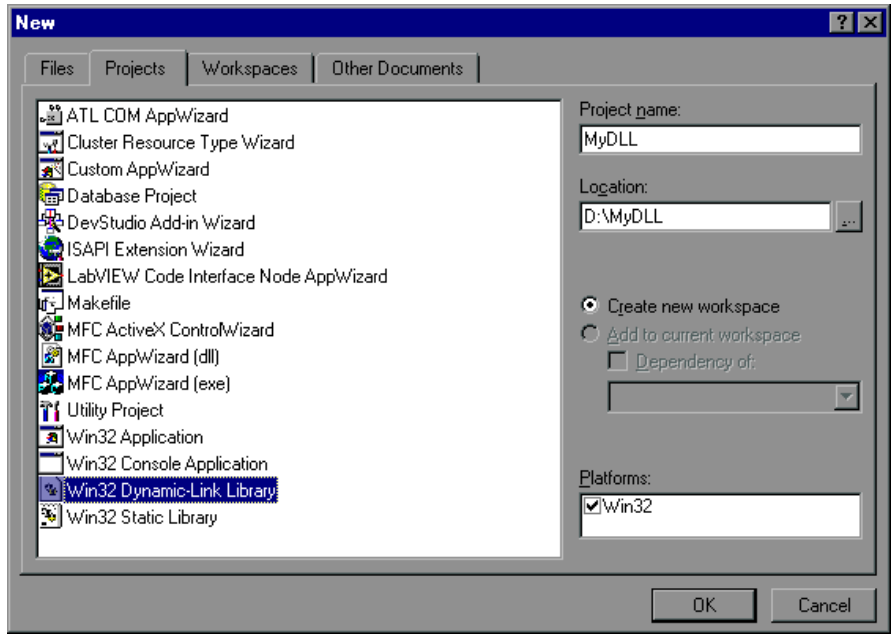



**Figure 2-2.** Creating a Project in Visual C++

3. The application prompts you to choose the type of DLL that you want to create; select **An empty DLL project**.
4. Click **Finish** to finish creating your project and return to the Visual C++ workspace.
5. From the **Project** menu, select **Add to Project»Files** 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 access the C++ tab of the **Project Settings** dialog box and make the following settings:
   a. Select **Code Generation** in the **Category** pull-down menu.
   b. For this example and for all configurations, set the **Struct member alignment** control to **1 Byte**.
   c. Select **Debug Multithreaded DLL** in the **Use run-time library** control to apply the Win32 Debug configuration, as shown in the following figure.

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

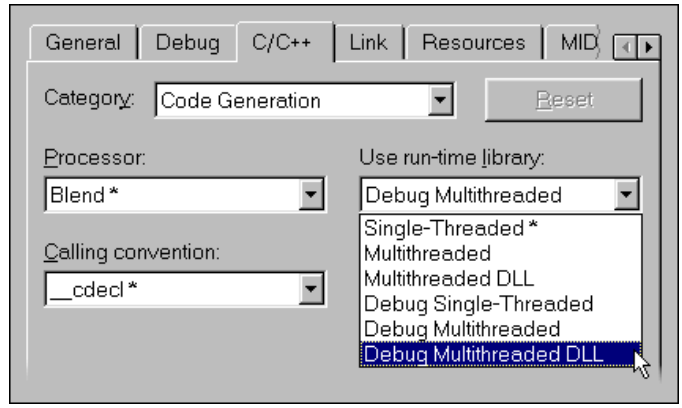



**Figure 2-3.** Setting the **Use run-time library** control, Microsoft Visual C++

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, Linux, or HP-UX

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, Linux, `.sl` on HP-UX.

**Note** If you use g++ to compile a shared library on HP-UX, check to be sure that the Dynamic Loader is calling the shared static global shared class initializers in that library.

## 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 Power Macintosh

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.

# Calling External APIs

It is frequently desirable to access external APIs from within LabVIEW code. Most often, a LabVIEW programmer accesses external APIs to obtain functionality that the operating system provides. Normally, you can use the LabVIEW Call Library Function object to accomplish this goal. You must provide the following information to the Call Library Function.

- 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

The function reference documentation for any API should provide most of the information that Call Library Function 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. In LabVIEW 6.0, 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.

## 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** in the **Type** control 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 object. 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.

## 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 may also be strings or other values. To identify all constants, inspect the header file for the API to find the definitions. The definition may 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 example describes how to complete an averaging VI called Array Average in which the LabVIEW Call Library Function calls myshared.dll. (In UNIX the shared library file has a .so or .sl extension.) The section *Building a Shared Library (DLL)* describes how to begin building the Array Average VI and how to create myshared.dll. This section describes the three stages for completing the Array Average VI so that it can call the avg_num function in myshared.dll.

- Complete configuration of the Call Library Function icon.
- Create the front panel.
- Create the block diagram.

## Configuration of Call Library Function

Complete the configuration of the Call Library Function object as follows.

1. If necessary, create an Array Average VI as described in the *Task 1: Build the Function Prototype in LabVIEW* section.
2. In the block diagram of the Array Average VI, right-click the Call Library Function icon and select **Configure** in the shortcut menu to invoke the **Call Library Function** dialog box.
3. For the **Library Name or Path** control, browse and select myshared.dll as the shared library that Call Library Function calls.

## Create Front Panel

Create the front panel of the VI as follows.

1. Place an array control, `Array`, to contain a scalar array of SGL with four members.
2. Place a numeric SGL indicator, `Average Value`, to display the result of your averaging calculation.
3. Place a numeric indicator, `Error`, to display any errors that your VI generates.

## Create the Block Diagram

Perform the following steps to complete the block diagram.

1. Connect the icons for following front panel controls to the Call Library Function icon.
   a. Connect the **Array** of data control to the `a` input.
   b. Connect the **Array Size** control to the `size` input.
   c. Connect a constant, zero, to the `avg` input.
   d. Connect the **Average Value** indicator to the `avg` output.
   e. Connect the **Error** indicator to the `error` output.
2. In the front panel, add dummy values to the array and run the VI to calculate the average of those values.
3. 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

LabVIEW users frequently want to access an API associated with hardware that they have purchased. With National Instruments hardware, however, you do not need to use the shared library object to gain access; all National Instruments products come with LabVIEW interfaces.

In this example you call a hypothetical interface card for a databus called “X-bus.” The X-bus interface card comes with a software driver for your operating system. The X-bus documentation provides standard information:

- A listing of all functions that 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`. 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 this information, you can configure the LabVIEW Call Library Function appropriately, as follows:

1. Create a new VI called Read Data and place a Call Library Function object in the Block diagram.
2. Right-click the Call Library Function object and select **Configure** in the shortcut menu.
3. In the **Call Library Function** dialog box, make the following settings.
   a. Select **stdcall (WINAPI)** in the **Calling Conventions** control.
   b. Type `XbusRead16`, in the Function Name control.
   c. Select **Signed 32 bit Integer** in the **Data Type** control for the **return type** parameter.
   d. Add a parameter and name it **offset** and select **Unsigned 32 bit Integer** in the **Data Type** control.
   e. Add a parameter and name it **data** and set its data type to be pointer to a signed 16-bit integer.
4. Inspect the function prototype that appears in the **Function Prototype** indicator. If this 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.

The following graphic shows what the front panel and block diagram of the final VI that calls `xbus.dll` might look like.

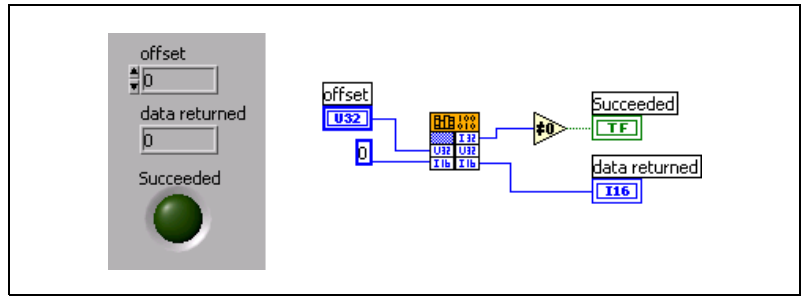



**Figure 2-4.** VI That Calls Hardware

## Example 3: Call the Win32 API

LabVIEW users frequently want to access the 32-bit Windows platform API (the 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 can also access this information at the Microsoft 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 like `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. The data types in the preceding prototype correspond to the following 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 |

In order 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. The preceding table shows HWND to be 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 an unsigned 32-bit integer, or, in LabVIEW terms, a U32. It is very common to run across handles like HWND 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—whose names always begin with the letter H in the Win32 API—as U32.

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.** Mapping Win32 Data Types to LabVIEW Data Types

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

## Constants

This section presents methods for finding the numerical values of constants in the Win32 API, using MessageBox constants as examples. The following table 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 98/95, 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 this 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, and 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.

**Tip** 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.

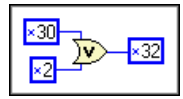



**Figure 2-5.** Combining Function Constants in LabVIEW

## Determining the Proper Library and Function Name

Before you can configure this 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 95 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 filename, but has a .dll extension instead of a .lib extension. This DLL that 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 one string argument. 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
```

This 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, there is a function `MessageBoxA` and a function `MessageBoxW`. In most cases in LabVIEW, a VI programmer uses 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 to call the `MessageBox` function. Remember that you must use the Standard (`__stdcall`) calling convention in calls to any function in the Windows SDK.

The following graphic shows a correctly configured instance of the Call Library Function. 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 for a separate example that teaches you how to configure controls in Call Library Function.

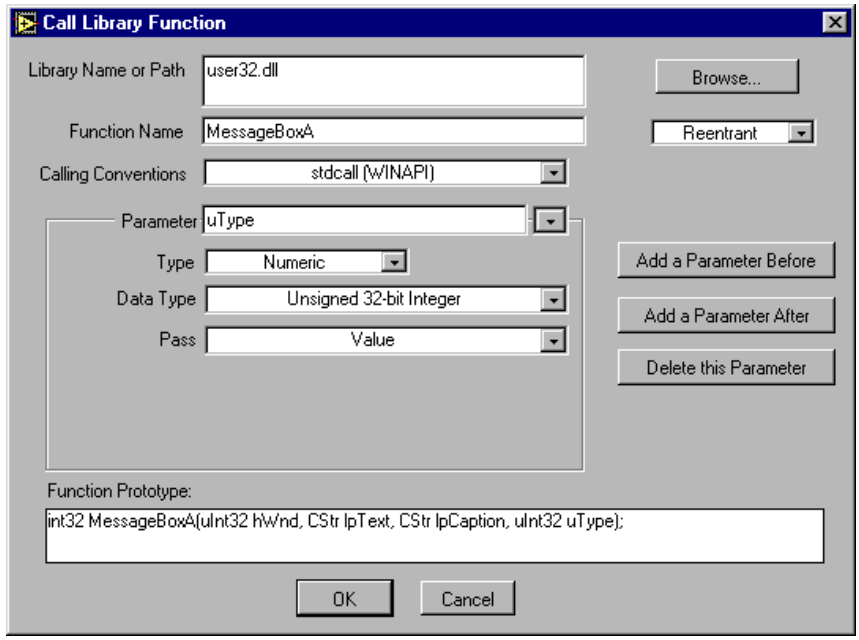



**Figure 2-6.** Configuring Call Library Function to call the Win32 API

You can configure the block diagram of this VI to match the following graphic.

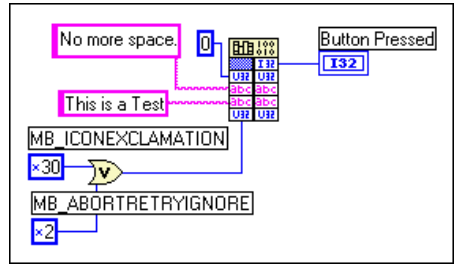



**Figure 2-7.** Block Diagram for a Call to the Win32 API

This VI generates the following message box.


**Figure 2-8.** Running a LabVIEW Call to the Win32 API

# Additional Examples of LabVIEW Calls to DLLs

You can access several other examples to learn more about calling DLLs from LabVIEW.

- If you have a sound card with Windows sound drivers installed on your system, investigate the Play Sound VI found in the LabVIEW Examples directory:

  `\LABVIEW\EXAMPLES\DLL\SOUND\PLAYSND.LLB\Play Sound.vi`

  You can use this VI to play Windows `.WAV` sound files on your computer from LabVIEW.

- 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:

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

- The LabVIEW example VI Hostname returns the host name of your computer, demonstrating how to use LabVIEW string handles:

  `\LABVIEW\EXAMPLES\DLL\HOSTNAME\hostname.vi`

- You can programmatically position your cursor anywhere on your monitor using the `SetCursorPos` function in `User32.DLL`. The function prototype is:

  ```
  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 Call Library Function in LabVIEW.

## Troubleshooting the Call Library Function

When your LabVIEW calls to DLLs generate errors, check for the following problems in your use of Call Library Function. Also refer to the *Troubleshooting your DLL* section and the *Troubleshooting Checklist* section.

- Make sure that the path to the DLL file is correct.
- If LabVIEW gives you the error message `function not found in library`, double-check your spelling of the name of the function you wish to call. Remember that function names are case sensitive. Also, be sure that your compiler has not decorated the function, as discussed in the *Preventing C++ Name Decoration* section.
- 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`. Also confirm that you are using the correct calling convention `__stdcall` or C.

## Troubleshooting your DLL

When LabVIEW calls to DLLs generate errors, check for the following problems in your DLL. Also refer to the *Troubleshooting the Call Library Function* section and the *Troubleshooting Checklist* section.

- 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, and so, the actual function name would be 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 may 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 use 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 wish to export in your header file with the `extern "C"` statement:

```
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. For more information about debugging, please refer to the appropriate manual for your compiler.
- Calling the DLL from another C program is also an excellent way to debug your DLL. By doing this, you have a means of testing your DLL independent of LabVIEW, thus helping you to identify any problems, sooner.

## Troubleshooting Checklist

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

- ❑ Call Library Function uses the proper calling convention (C or `__stdcall`).
- ❑ Call Library Function has the correct path to the DLL.
- ❑ Call Library Function 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.
- ❑ In the Call Library Function icon, data is wired to the input terminals of all the parameters that you are passing to a DLL function. Also, check that the function is properly configured for all input parameters.

- Return types and data types of arguments for functions in Call Library Function exactly match the data types your function uses. Erroneous data type assignments can cause crashes.
- Call Library Function 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 Call Library Function passes a LabVIEW Array Handle or LabVIEW String Handle, and,
  - *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 may result in a crash by overwriting other data stored in LabVIEW memory.

- Call Library Function 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.
- 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 may return `NULL` values as part of the data string.
- For arrays or strings of data, you *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`, you 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, you export functions with the `extern "C"{}` statement in your header file in order to prevent name mangling.
- ❑ For a DLL that you have written, you 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.
- ❑ You tested 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 will 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 configure LabVIEW to use the C calling convention in the `.c` source file you build with the LabVIEW Call Library Function. 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 module definition (`.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.

```
LIBRARY myshared
   EXPORTS
      avg_num
```

The preceding code example demonstrates 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 Call Library Function.

## 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.

The following list presents the characteristics of Array Data Pointers, whether you pass them in the Windows API or in another API. 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.

- 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.
- Specifically, *never* resize an array or perform operations that may change the length of the array data passed from LabVIEW. Resizing may cause a crash because the pointer sent is not an allocated block, but rather, 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 it 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.

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.

- For CodeWarrior, include `labview.export.stub`.
- For Symantec, include `labview.sym.lib`.
- For Visual C++, include `labview.lib`.

## String Data

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

- **C String Pointer**—Pointer to the string, followed by a NULL character. Most Win32 API functions use this C-style string pointer.
- **Pascal String Pointer**—Pointer to the string, preceded by a length byte.
- **LabVIEW String Handle**—Pointer to a pointer to the string, preceded by four bytes of length information.
- **LabVIEW String Handle Pointer**—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 signed 32-bit integer that stores how many characters appear in the string. Thus, a string with *n* characters will require *n* + 4 bytes to store in memory. For example, in the following graphic the string `text` contains four characters. When LabVIEW stores the string, the first four bytes contain the value `4` as a signed 32-bit 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 the following figure. If you pass a LabVIEW String Handle from Call Library Function to the DLL, then you can use the LabVIEW CIN functions like `DSSetHandleSize` to resize the LabVIEW String Handle.

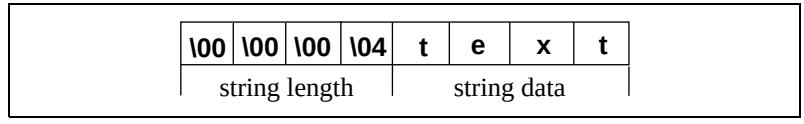



**Figure 2-9.** The LabVIEW String Format

Remember, you must add `labview.lib` to a Visual C++ project, `labview.export.stub` to a CodeWarrior project, and `labview.sym.lib` to a Symantec project.

The Pascal string format is nearly identical to the LabVIEW string format, but instead of storing the length of the string as a signed 32-bit integer, it is stored as an unsigned 8-bit integer. This limits the length of a Pascal style string to 255 characters. A graphical representation of a Pascal string appears in the following figure. A Pascal string that is *n* characters long will require *n* + 1 bytes of memory to store.

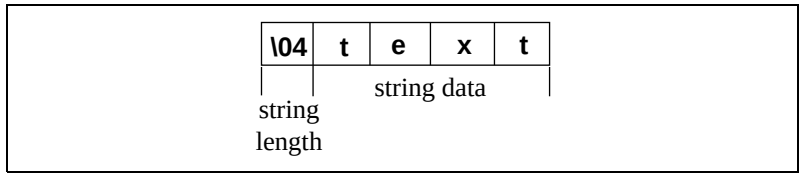



**Figure 2-10.** The Pascal String Format

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 one observes that C strings are declared as `char *`, where `char` is typically an unsigned byte. 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. 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. For binary data where NULLs may be present, consider an array of unsigned 8-bit integers. If you treat the string as a C-style string, your program will incorrectly assume that the end of the string has been reached, when in fact your instrument is returning a numeric value of zero. An illustration of how a C-style string is stored in memory appears in the following figure.

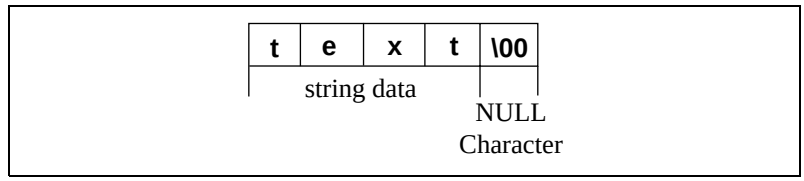



**Figure 2-11.** The C String Format

When you pass string data to a DLL, you must follow the same guidelines as for arrays:

- *Never* resize a string, concatenate a string, or perform operations that may 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, you should first 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 Call Library Function to the DLL, then you can use the LabVIEW CIN functions like `DSSetHandleSize` to resize the LabVIEW string handle.

**Note** To use the LabVIEW CIN function calls you must add `labview.lib` to a Visual C++ project, `labview.export.stub` to a CodeWarrior project, and `labview.sym.lib` to a Symantec project.

## Array and String Tip

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

1. Determine the number of elements that the array requires, or the length of the string to be returned. Have this first function return the desired size to LabVIEW.
2. In LabVIEW, initialize an array or string with default values, and pass this array to a second function in your DLL, which actually places the data into the array. If you are working with string-based instrument control, it may 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 object to your DLL, you can use the LabVIEW CIN functions to resize or create an array or string. Refer to the *Required Libraries* section for more information about this set of functions.

# 3

# CINs

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

## 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.

### Macintosh

CINs in LabVIEW for Macintosh access a shared libraries. 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 come with LabVIEW.

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

LabVIEW header files are compatible with these two environments. Header files might need modification for other environments.

### Microsoft Windows

LabVIEW for Windows supports CINs created with any of the following compilers:

- Microsoft Visual C++
- Symantec C

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

### Solaris, Linux, and HP-UX

LabVIEW for Sun supports external code compiled in a shared library format. To prepare this library for LabVIEW, use LabVIEW utility `lvsbutil`.

The gcc compiler is tested thoroughly with LabVIEW on Solaris, Linux, and HP-UX platforms. For Solaris, Sun Workshop C Compiler is also tested thoroughly with LabVIEW.

## Resolving Multithreading Issues

You must resolve two issues in order 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. Perform the following steps to make LabVIEW recognize a CIN node as thread safe.

Add the `CINProperties` function to your CIN code, in the prototypes section of your `.c` source file:

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

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 list presents basic answers to the question, *Is my CIN code thread safe?*

- The CIN code is thread safe when it stores no unprotected global data (for example, no global variables, no files on disk, and so on); does not access any hardware (in other words, does not contain register-level programming); and makes no calls to any functions, shared libraries, or drivers that are not thread safe.
- The CIN 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; and the code accesses no global resources through CIN housekeeping routines, such as, `CINInit`, `CINAbort`, `CINDispose`, and others.

# Creating a CIN

In general, to create a CIN, describe in LabVIEW the data you want to pass to the CIN. Then, write the code for the CIN using one of the supported programming languages. After you compile the code, run a utility that puts the compiled code into a format LabVIEW can use. Then, instruct LabVIEW to load the CIN.

If you run the VI at this point 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` directory contains a `CINs` directory that includes all of the examples given in this manual. The names of the directories in `CINs` correspond to the CIN name in the examples.

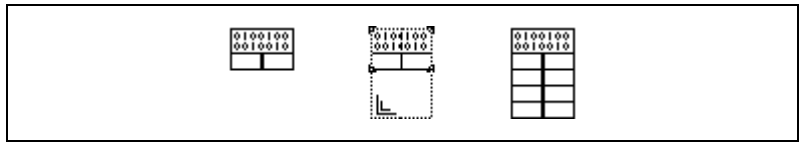
To create a CIN, complete the following steps.

## Step 1. Set Up Input and Output Terminals for a CIN

Access the Code Interface Node located on the **Functions»Advanced** palette and place it 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, resize the node, then right-click the node and select **Add Parameter**.

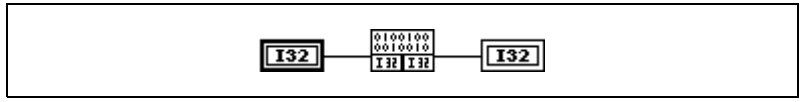
The following illustration shows how to resize the node to add parameters.



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.

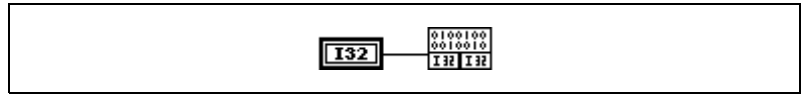
### Input-Output Terminals

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




When the VI calls the CIN, the only argument LabVIEW passes to the CIN object code is a pointer to the value of the 32-bit integer input. When the CIN completes, LabVIEW then passes the value referenced by the pointer to the 32-bit integer indicator. When you wire controls and indicators to the input and the output terminals of a terminal pair, LabVIEW assumes 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.

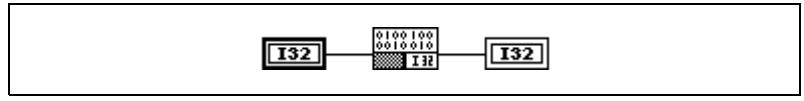
Consider the same CIN, but with no indicator wired to the output terminal, as shown in the following illustration.




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

## Output-Only Terminals

If you use a terminal pair only to return a value, make it an output-only terminal pair by resizing the node then right-clicking the node and selecting **Output Only**. If a terminal pair is output-only, the input terminal is gray, as shown in the following illustration.




For output-only terminals, LabVIEW creates storage space for a return value and passes the value by reference to the CIN the same way 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 left (input) terminal of the terminal pair as shown in the previous illustration. In this case, the output terminal takes on the same data type as the input terminal. LabVIEW uses the input type only to determine the data type for the output terminal; the CIN does not use or affect the data of the input wire.

To remove a pair of terminals from a CIN, right-click the terminal you want to remove and select **Remove Terminal**. 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.

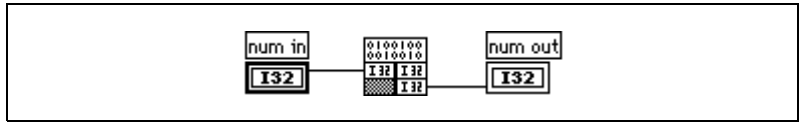
## Step 2. Wire the Inputs and Outputs to the CIN

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 in 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 node and select **Create .c File** 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, consider the following call to a CIN, which takes a 32-bit integer as an input and returns a 32-bit integer as an output.




The following code excerpt is the initial `.c` file for this node. You can write eight routines for the CIN. 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.

```
/*
 * CIN source file
 */

#include "extcode.h"

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

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

    /* ENTER YOUR CODE HERE */
    return noErr;
    }
```

This `.c` file is a template in which you must write C code. `extcode.h` 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 may conflict with the definitions of system header files. The `cintools` directory also contains `hosttype.h`, which resolves these differences. This header file 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 by `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, 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. To do so, write your `CINLoad` routine as follows:

```
CIN MgErr CINLoad(RsrcFile reserved) {
       Unused (reserved);
       /* ENTER YOUR 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 code shows an example of how to fill out the `CINRun` routine from the previously shown LabVIEW-generated `.c` file to multiply a number by two. Refer to the *Passing Parameters* section in Chapter 4, *Programming Issues for CINs*, for information about how LabVIEW passes data to a CIN, with several examples.

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

## 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. These tools vary depending 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.

### Compile on Macintosh

LabVIEW for Macintosh 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.

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

Always use the latest Universal headers containing definitions for Power Macintosh compilers.

#### Metrowerks CodeWarrior for Power Macintosh

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 Power Macintosh, 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 this file 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.

To build the CIN, select **Project»Make**.

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

**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.

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 Power Macintosh. Several scripts are available for the MPW environment to help you build CINs.

- CINMake—This script uses a simplified form of a makefile you provide. You can run it every time you need to rebuild your CIN.
- LVMakeMake—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*

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

  Type of external code you want to create. For CINs, use a type of CIN.
- codeDir = *codeDir:*

  Complete pathname to the folder containing the .c file used for the CIN.
- cinToolsDir = *cinToolsDir:*

  Complete pathname to the LabVIEW cintools:MPW folder.
- LVMVers = 2

  Version of CINMake script reading this .lvm file.
- inclDir = *-i inclDir:*

  (Optional) Complete or partial pathname to a folder containing any additional .h files.

- `otherPPCObjFiles = otherPPCObjFiles`

  (Optional) List of additional object files (files with a `.o` extension) your code needs to compile. Separate the names of files with spaces.
- `ShLibs = sharedLibraryNames`

  (Optional) 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`

  (Optional) 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:

  ```
  -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`.

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

Go to the MPW Worksheet and enter the following commands. Set your current folder to the CIN folder:

`Directory <pathname to directory of your CIN>`

Run the LabVIEW `CINMake` script:

`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 follows, 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 | |
| -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 follows, 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 Power Macintosh makefile for a CIN named myCIN, use the following command:

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

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

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

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++ Command Line

This section describes using command line tools in Windows 2000/NT/9*x* to build CINs.

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

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

   ```
   CINTOOLSDIR = c:\labview\cintools
   ```

   **(Windows 9*x*)** 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 CIN or external subroutine (for example, `mult`) .
   - `type` is CIN or LVSB, depending on whether it is a CIN or an external subroutine.
   - `!include $(CINTOOLSDIR)\ntlvsb.mak`

   To define additional include paths for a CIN you must add a `CINCLUDES` line to the `.lvm` file, as follows:

   ```
   CINCLUDE = -Ipathnames
   ```

   You must include the `-I` argument on the line and `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.

## Visual C++ IDE

To build CINs using the Visual C++ Integrated Development Environment, complete the following steps.

1. Create a new DLL project. Select **File»New** and select **Win32 Dynamic-Link Library** as the project type. You can name your project whatever you want.
2. Add CIN objects and libraries to the project. Select **Project»Add To Project»Files** and select `cin.obj`, `labview.lib`, `lvsb.lib`, and `lvsbmain.def` from the `Cintools\Win32` subdirectory. You need these files to build a CIN.
3. Add `Cintools` to the include path. Select **Project»Settings** and change **Settings for** to **All Configurations**. Select the **C/C++** tab and set the category to **Preprocessor**. Add the path to your `Cintools` directory in the **Additional include directories** field.
4. Set alignment to **1 byte**. Select **Project»Settings** and change **Settings For** to **All Configurations**. Select the **C/C++** tab and set the category to **Code Generation**. Select the **Struct member alignment** tab and select **1 byte**.
5. 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.
6. 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 **Build commands** field as follows; this code should appear on a single line:

   ```
   "<your path to cintools>\win32\lvsbutil" $(TargetName) -d
   "$(WkspDir)\$(OutDir)"
   ```

   Change Output file fields to `$(OutDir)$(TargetName).lsb`.

**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.

### 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`.

The gcc compiler and the Sun Workshop C Compiler are the only compilers tested thoroughly with LabVIEW.

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

Existing Solaris 1.*x* and 2.*x* (for LabVIEW 3.0) CINs do not operate correctly if they reference functions not in the System V Interface Definition (SVID) for `libc`, `libsys`, and `libnsl`. Recompile your existing CINs using the shared library format to make sure your CINs function as expected.

## HP-UX and Linux

The gcc compiler is the only compiler tested with LabVIEW.

## 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 follows, with optional parameters listed in brackets.

`lvmkmf [-o Makefile] LVSBName`

*LVSBName* is the name of the CIN or external subroutine 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 makefile produced assumes the `cin.o`, `libcin.a`, `makeglue`*`XXX`*`.awk`, and `lvsbutil` files are in certain locations, where *`XXX`* is `SVR4` on Solaris 2.*x*, `linux` on Linux, and `HP` on HP-UX. 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**. Select the `.lsb` file you created in *Step 4. Compile the CIN Source Code*.

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** option. The file containing the object code for the CIN must have an extension of `.lsb`.

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

## LabVIEW Manager Routines

LabVIEW has a suite of routines that 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 three categories: memory manager, file manager, and 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 two 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.

## Pointers as Parameters

Some manager functions have a parameter that is a *pointer*. These parameter type descriptions are identified by a trailing asterisk (such as the **hp** parameter of the `AZHandToHand` memory manager function) 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:

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

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

Correct example:

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

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

Incorrect example:

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

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

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

In the incorrect example, `p` is a pointer to a Pascal string, but the pointer is not initialized to point to any allocated buffer. `FNamePtr` expects its caller to pass a pointer to an allocated space, and writes the name of the file referred to by `path` into that space. Even if the pointer does not point to a valid place, `FNamePtr` 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.

# 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 or external subroutine.

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. To debug CINs in Windows, complete the following steps.

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 9*x* has a single method of setting a debugger trap, while Windows 2000/NT can use the Windows 95 method or another.

   The method common to Windows is to insert a debugger break using an in-line assembly command:

   ```
   _asm int 3;
   ```

Adding this to `CINLoad` gives you the following:

```
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, you can use the `DebugBreak` function. This function exists in Windows 9*x*, 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:

```
#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:

   ```
   DEGUG = 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. Type 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 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 can then 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

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, called managers, which you can use in external code modules. These include the memory manager, the file manager, and the support manager.

## Passing Parameters

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

### Parameters in the CIN .c File

When you right-click a CIN on a block diagram and select **Create .c File**, LabVIEW creates a `.c` file in which you can enter your CIN code. The `CINRun` function and its prototype are given, and its parameters correspond to the data types being passed to the CIN in the block diagram. Refer to the *CIN Routines* section in Chapter 5, *Advanced Applications*, for more information about CIN routines (`CINInit`, `CINLoad`, and so on).

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—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 explicit about data type size, such as `int16`, `int32`, `float32`, and so on. LabVIEW includes a header file, `extcode.h`, that contains typedefs associating these LabVIEW data types with the corresponding data type for the supported compilers of each platform.

`extcode.h` defines some constants and types whose definitions may conflict with the definitions of system header files. The LabVIEW `cintools` directory also contains the `hosttype.h` file, which resolves these differences. This header file 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 by `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 an `int32` by reference. `extcode.h` typedefs an `int32` to the appropriate data type for the compiler you use (if it is a supported compiler). Therefore, you can use the `int32` data type in external code you write.

## Passing Fixed-Size Data to CINs

As described in the *Creating a CIN* section in 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. LabVIEW passes parameters to the `CINRun` routines in the same order as you wire data to the CIN—the first terminal pair corresponds to the first parameter, and the last terminal pair corresponds to the last parameter.

Refer to the following sections for information about how LabVIEW passes fixed-size parameters to CINs. Refer to the *Passing Variably Sized Data to CINs* section in this chapter for information about manipulating variably sized data, such as arrays and strings.

### Scalar Numerics

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

### 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, it 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, it 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 component is another cluster, LabVIEW stores this cluster value as a component of the main cluster.

## Return Value for CIN Routines

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

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

The LabVIEW header file `extcode.h` defines the word CIN to be either Pascal or nothing, depending on the platform. Prefacing a function with the word `Pascal` causes some C compilers to use Pascal calling conventions instead of C calling conventions to generate the code for the routine.

LabVIEW uses 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. This behavior enables LabVIEW to handle conditions when a VI runs out of memory. By aborting the running VI, LabVIEW can possibly free enough memory to continue running correctly.

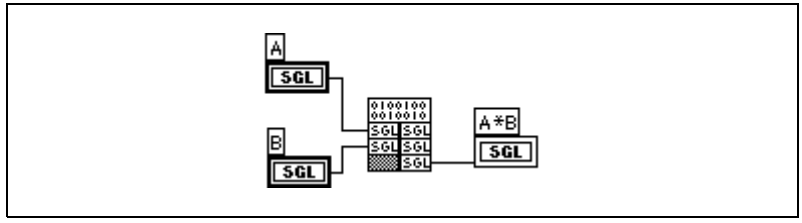
## Examples with Scalars

The following examples 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

To create a CIN that takes two single-precision floating-point numbers and returns their product, complete the following steps.

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 a front panel. Wire the node as shown in the following illustration. **A*B** is wired to an output-only terminal pair.




4. Save the VI as `mult.vi`.
5. Right-click the node 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;
    }
```

This .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 you wired them (top to bottom) to the CIN. Thus, A, B, and A_B are pointers to **A**, **B**, and **A*B**, respectively.

As described in the *Parameters in the CIN .c File* section earlier in this chapter, 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 may not be true in all cases, because the C standard does not define the sizes for the various data types. You can use these LabVIEW data types in your code because extcode.h associates these data types with the corresponding C data type for the compiler you are using. In addition to defining LabVIEW data types, extcode.h also prototypes LabVIEW routines you can access. Refer to the *Manager Overview* section for descriptions of these data types and routines.

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

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

CINRun multiplies the values to which A and B refer and stores the results in the location to which A_B refers. It is important 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 *Compile on Macintosh* section in 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`

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

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

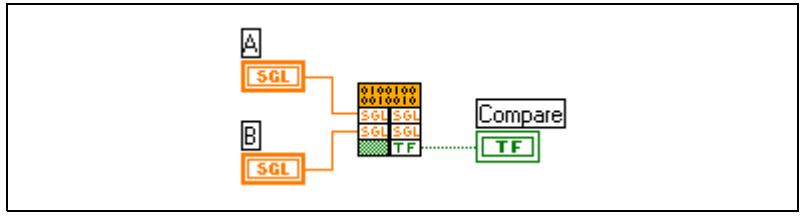
You should be able to run the VI. If you save the VI, LabVIEW saves the CIN object code along with the VI.

## Comparing Two Numbers, Producing a Boolean Scalar

To create a CIN that compares two single-precision numbers, complete the following steps. If the first number is greater than the second one, the return value is TRUE; otherwise, the return value is FALSE. This example shows only the block diagram and the code.

1. To create the CIN, follow the instructions in the *Creating a CIN* section in Chapter 3, *CINs*.

   The diagram for this CIN is shown in the following illustration.




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;
    }
```

# 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. In this case, LabVIEW might have to move the data to a location that offers more space.

To accommodate this relocation of memory, LabVIEW uses handles to refer to the storage location of variably sized data. A handle is a pointer to a pointer to the desired data. LabVIEW uses handles instead of simple pointers because handles allow LabVIEW to move the data without invalidating references from your code to the data. If LabVIEW moves the data, LabVIEW updates the intermediate pointer to reflect the new location. If you use the handle, references to the data go through the intermediate pointer, which always reflects the correct location of the data. Refer to the *Using Pointers and Handles* section later in this chapter for more 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` or, if the data is stored in the application zone, the `AZSetHandleSize` routine. Both routines work, but 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 *XX*`SetHandleSize`, 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 in this chapter for a description of this function.

If you are not familiar with alignment differences for various platforms, the following examples highlight the problem. `SetCINArraySize` and `NumericArrayResize` solve these problems.

- In Windows, a one-dimensional array of double-precision floating-point numbers is stored in a handle, and the first four bytes describe the number of elements in the array. These four bytes are followed by the 8-byte elements that make up the array. In Solaris, 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 eight-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.

## Arrays and Strings

LabVIEW passes arrays by handle, as described in the *Alignment Considerations* section earlier in this chapter. 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, and the second four bytes indicate the number of columns. These dimension fields can be followed by filler and then the actual data. Each element can also have padding to meet alignment requirements.

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

**Note** LabVIEW 4.*x* stored Boolean arrays in memory as a series of bits packed to the nearest 16-bit word. LabVIEW 4.*x* ignored unused bits in the last word. 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 Refer to Chapter 6, *Function Descriptions*, for more 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 this cluster value as a component of the main cluster. If a component is an array or string, LabVIEW stores a handle to the array or string component in the structure.

## Resizing Arrays and Strings

To resize return arrays and strings you pass to a CIN, use the LabVIEW `SetCINArraySize` routine. Pass to the function the handle you want to resize, information describing the data structure, and the desired size of the array or handle. The function takes into account any padding and alignment needed for the data structure. However, the function 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 this function the array you want to resize, a description of the data structure, and information about the new size of the array.

When you resize arrays of variably sized data (for example, arrays of strings) with the `SetCINArraySize` or `NumericArrayResize` routines, consider the following issues:

- If the new size of the array is smaller, LabVIEW disposes of the handles used by the disposed element. Neither function sets the dimension field of the array. You must do this in your code after the function call.
- If the new size 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.

The following sections describe the `SetCINArraySize` and `NumericArrayResize` routines.

## SetCINArraySize

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

### Purpose

`SetCINArraySize` resizes a data handle based on the data structure of an argument you pass to the CIN. It 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. The leftmost parameter has a parameter number of 0, and the rightmost has a parameter number of $n - 1$, where $n$ is the total number of parameters. |
| **newNumElmts** | `int32` | New number of elements to which the handle should refer. For a one-dimensional array of five values, pass a value of 5. For a two-dimensional array of two rows by three columns, pass a value of 6. |

### Return Value

`MgErr`, which can contain the errors in the following list. Refer to the *Manager Overview* section later in this chapter for more information about `MgErr`.

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

## NumericArrayResize

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

### Purpose

`NumericArrayResize` resizes a data handle referring 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. The header file `extcode.h` defines the following constants for this argument:<br>`iB` Array of signed 8-bit integers<br>`iW` Array of signed 16-bit integers<br>`iL` Array of signed 32-bit integers<br>`uB` Array of unsigned 8-bit integers<br>`uW` Array of unsigned 16-bit integers<br>`uL` Array of unsigned 32-bit integers<br>`fS` Array of single-precision (32-bit) numbers<br>`fD` Array of double-precision (64-bit) numbers<br>`fX` Array of extended- precision numbers<br>`cS` Array of single-precision complex numbers<br>`cD` Array of double-precision complex numbers<br>`cX` Array of extended-precision complex numbers |
| **numDims** | `int32` | Number of dimensions in the data structure to which the handle refers. Thus, if the handle refers to a two-dimensional array, pass a value of 2. |

| Name | Type | Description |
|---|---|---|
| ***dataHP** | `UHandle` | Pointer to the handle you want to resize. If this is a pointer to `NULL`, LabVIEW allocates and sizes a new handle appropriately and returns the handle in ***dataHP**. |
| **totalNewSize** | `int32` | New number of elements to which the handle should refer. For a unidimensional array of five values, pass a value of 5. For a two-dimensional array of two rows by three columns, pass a value of 6. |

## Return Values

`MgErr`, which can contain the errors in the following list. Refer to the *Manager Overview* section later in this chapter for more information about `MgErr`.

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

# Examples with Variably Sized Data

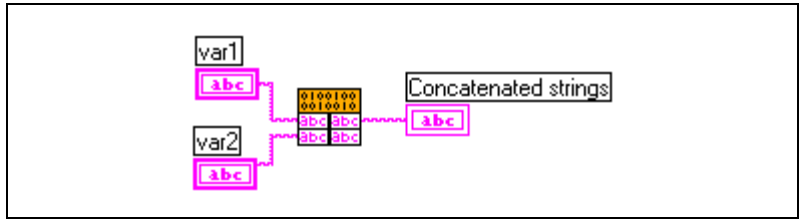
The following examples describe how to create CINs that work with variably sized data types. Refer to Chapter 3, *CINs*, for more information about creating CINs.

## Concatenating Two Strings

To create a CIN that concatenates two strings and use input-output terminals by passing the first string as an input-output parameter to the CIN, complete the following steps. The top right terminal of the CIN returns the result of the concatenation. This example shows only the diagram and the code.

1. To create the CIN, follow the instructions in the *Creating a CIN* section in Chapter 3, *CINs*.

   The diagram for this CIN is shown in the following illustration.




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, as follows:

```
CIN MgErr CINRun(
    LStrHandle strh1,
    LStrHandle strh2) {
    int32 size1, size2, newSize;
    MgErr err;
    size1 = LStrLen(*strh1);
    size2 = LStrLen(*strh2);
    newSize = size1 + size2;
    if(err = NumericArrayResize(uB, 1L,
    (UHandle*)&strh1, newSize))
    goto out;
    /* append the data from the second string to
    first string */
    MoveBlock(LStrBuf(*strh2),
    LStrBuf(*strh1)+size1, size2);
    /* update the dimension (length) of the
    first string */
    LStrLen(*strh1) = newSize;
out:
    return err;
    }
```

   In this example, CINRun is the only routine that performs substantial operations. CINRun concatenates the contents of strh2 to the end of strh1, with the resulting string stored in strh1.

5. 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 case, NumericArrayResize could fail if LabVIEW does not have enough memory to resize the string. Returning the error code gives LabVIEW a chance to handle the error. If CINRun reports an error, LabVIEW aborts the calling VIs. Aborting the VIs might free up enough memory so LabVIEW can continue running.

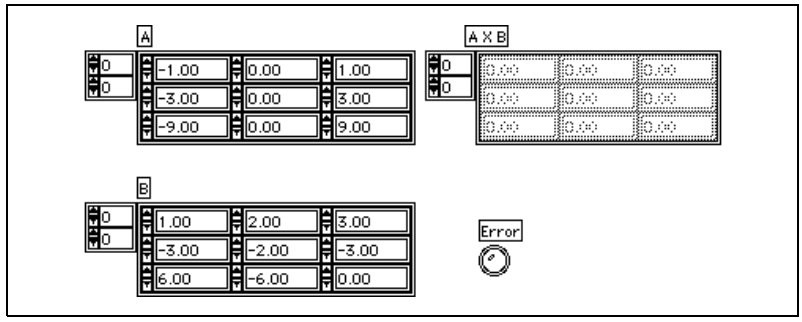
   After resizing the string handle, MoveBlock copies the second string to the end of the first string. MoveBlock is a support manager routine that moves blocks of data. Finally, this example sets the size of the first string to the length of the concatenated string.

## Computing the Cross Product of Two Two-Dimensional Arrays

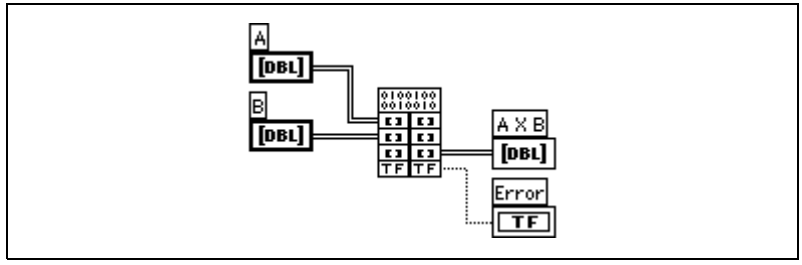
To create a CIN that accepts two two-dimensional arrays and then computes the cross product of the arrays, complete the following steps. The CIN returns the cross product in a third parameter and a Boolean value as a fourth parameter. This Boolean parameter is TRUE if the number of columns in the first 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.

1. To create the CIN, follow the instructions in the *Creating a CIN* section in Chapter 3, *CINs*.

   The front panel for this VI is shown in the following illustration.




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




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

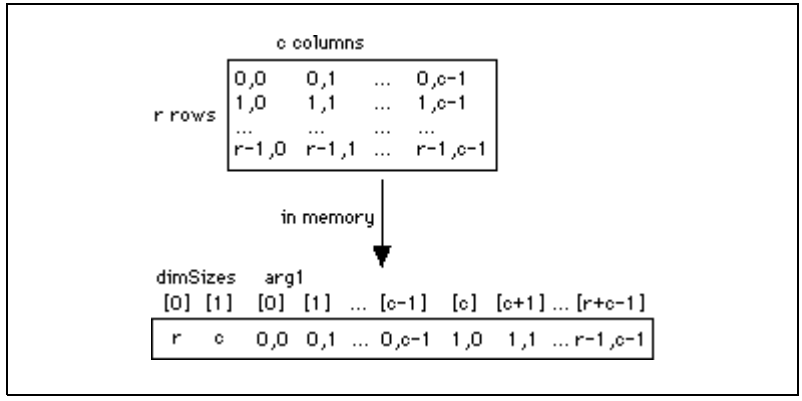
```
/*
 * CIN source file
 */
#include "extcode.h"
```

```
#define ParamNumber 2
   /* The return parameter is parameter 2 */
#define NumDimensions 2
   /* 2D Array */
/*
 * typedefs
 */
typedef struct {
   int32 dimSizes[2];
   float64 arg1[1];
   } TD1;
typedef TD1 **TD1Hdl;
CIN MgErr CINRun(TD1Hdl A, TD1Hdl B, TD1Hdl
AxB, 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 = (*ah)->dimSizes[1]) !=
   (*bh)->dimSizes[0]) {
      *error = LVTRUE;
      goto out;
      }
   *error = LVFALSE;
   rows = (*ah)->dimSizes[0];
      /* number of rows in a and result */
   cols = (*bh)->dimSizes[1];
      /* number of cols in b and result */
   newNumElmts = rows * cols;
   if (err = SetCINArraySize((UHandle)AxB,
         ParamNumber, newNumElmts))
      goto out;
   (*resulth)->dimSizes[0] = rows;
   (*resulth)->dimSizes[1] = cols;
   aElmtp = (*ah)->arg1;
   bElmtp = (*bh)->arg1;
   resultElmtp = (*result)->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*cos + 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 `result`. 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 the following illustration.




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

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

## Working with Clusters

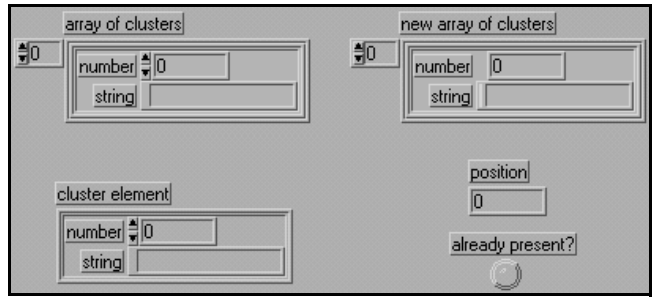
To take an array of clusters and a single cluster as inputs, complete the following steps. The clusters contain a signed 16-bit integer and a string. The input for the array of clusters is an input-output terminal. In addition to the array of clusters, the CIN returns a Boolean parameter and a signed

32-bit integer. If the cluster value is already present in the array of clusters, the CIN sets the Boolean parameter to TRUE and returns the position of the cluster in the array of clusters using the 32-bit integer output. If the cluster value is not present, the CIN adds it to the array, sets the Boolean output to FALSE, and returns the new position of the cluster in the array of clusters.

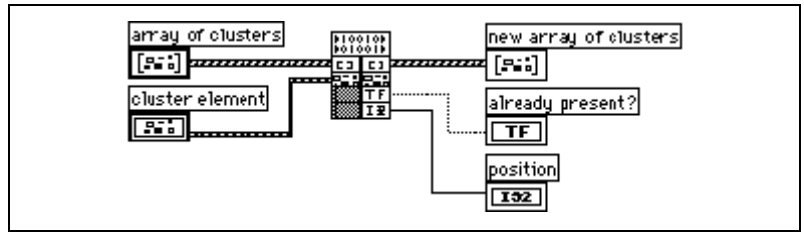
This example shows only the front panel, block diagram, and source code.

1. To create the CIN, follow the instructions in the *Creating a CIN* section in Chapter 3, *CINs*.

   The front panel for this VI is shown in the following illustration.




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




2. Save the VI as `tblsrch.vi`.
3. Save the `.c` file for the CIN as `tblsrch.c`. Following is the source code for `tblsrch.c` with the `CINRun` routine added.

```
/*
 * CIN source file
 */
#include "extcode.h"
#define ParamNumber 0
   /* The array parameter is parameter 0 */
/*
 * typedefs
 */
```

```
typedef struct {
   int16 number;
   LStrHandle string;
   } TD2;
typedef struct {
   int32 dimSize;
   TD2 arg1[1];
   } TD1;
typedef TD1 **TD1Hdl;
CIN MgErr CINRun(
                TD1Hdl       clusterTableh,
                TD2          *elementp,
                LVBoolean    *presentp,
                int32        *positionp);
CIN MgErr CINRun(
                TD1Hdl       clusterTableh,
                TD2          *elementp,
                LVBoolean    *presentp,
                int32        *positionp) {
   int32        size,i;
   MgErr        err=noErr;
   TD2          *tmpp;
   LStrHandle   newStringh;
   TD2          *newElementp;
   int32        newNumElements;

size = (*clusterTableh)->dimSize;
tmpp = (*clusterTableh)->arg1;
*positionp = -1;
*presentp = LVFALSE;
   for(i=0; i<size; i++) {
      if(tmpp->number == elementp->number)
         if(LStrCmp(*(tmpp->string),
                *(elementp->string)) == 0)
            break;
      tmpp++;
      }
   if(i<size) {
      *positionp = i;
      *presentp = LVTRUE;
      goto out;
      }
   newStringh = elementp->string;
```

```
    if(err = DSHandToHand((UHandle *)
    &newStringh))
        goto out;
    newNumElements = size+1;
    if(err =
        SetCINArraySize((UHandle)clusterTableh,
        ParamNumber,
        newNumElements)) {
        DSDisposeHandle(newStringh);
        goto out;
        }
    (*clusterTableh)->dimSize = size+1;
    newElementp = &((*clusterTableh)
    ->arg1[size]);
    newElementp->number = elementp->number;
    newElementp->string = newStringh;
    *positionp = size;
out:
    return err;
    }
```

In this example, `CINRun` is the only routine performing substantial operations. `CINRun` first searches through the table to see if the element is present. `CINRun` then compares string components using the LabVIEW routine `LStrCmp`, which is described in Chapter 6, *Function Descriptions*. If `CINRun` finds the element, the routine returns the position of the element in the array.

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

# 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 these

routines, you can create external code that works on all platforms that LabVIEW supports.

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, it is almost impossible to create code that works identically across multiple compilers.

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

Most applications need routines for allocating and deallocating memory on request. You can use the *memory manager* to dynamically allocate, manipulate, and release memory. The LabVIEW memory manager supports dynamic allocation of both non-relocatable and relocatable blocks, using pointers and handles. Refer to the *Memory Manager* section later in this chapter for more information.

Applications that manipulate files can use the functions in the *file manager*. This set of routines supports basic file operations such as creating, opening, and closing files, writing data to files, and reading data from files. In addition, 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 pathnames (`Paths`) that indicates a file or directory path independent of the platform. You can translate a `Path` to and from a host platform's conventional format for describing a file pathname. Refer to the *File Manager* section later in this chapter for more information.

The *support manager* contains a collection of generally useful functions, such as functions for bit or byte manipulation of data, string manipulation, mathematical operations, sorting, searching, and determining the current time and date. Refer to the *Support Manager* section later in this chapter for more information.

# Basic Data Types

Manager data types include five basic data types: scalar, char, dynamic, memory-related, and 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 form is called a `Bool32`, and it is stored as a 32-bit value.

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

The following table describes the two forms of Boolean scalars.

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

### Numeric

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

- Signed integers
  - `int8` 8-bit integer
  - `int16` 16-bit integer
  - `int32` 32-bit integer
- Unsigned integers
  - `uInt8` 8-bit unsigned integer
  - `uInt16` 16-bit unsigned integer

  - `uInt32` 32-bit unsigned integer
- Floating-point numbers
  - `float32` 32-bit floating-point number
  - `float64` 64-bit floating-point number
  - `floatExt` extended-precision floating-point number

In Windows, extended-precision numbers are stored as an 80-bit structure with two `int32` components, `mhi` and `mlo`, and an `int16` component, `e`. In Sun, extended-precision numbers are stored as 128-bit floating-point numbers. In Power Macintosh, extended-precision numbers are stored in the 128-bit double-double format. In HP and Concurrent, extended-precision numbers are the 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. The following code gives the type definitions for each of these complex data types.

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

## char

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

```
typedef uInt8 uChar;
```

## Dynamic

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 earlier in 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. This gives 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, and the specified number of characters follow. This is the string data type used by LabVIEW block diagrams. The following code is the type definition for an `LStr` string.

```
typedef struct {
   int32 cnt;
   /* number of bytes that follow */
```

```
uChar str[1];
/* cnt bytes */
} LStr, *LStrPtr, **LStrHandle;
```

### Concatenated Pascal String (CPStr)

Many algorithms require manipulation of lists of strings. Arrays of strings are usually the most convenient representation for lists. However, this representation 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. You can use support manager routines to create and manipulate lists using this data structure.

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. These data types have the following type definitions.

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

Refer to Chapter 6, *Function Descriptions*, for more 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

The memory manager is a set of platform-independent routines you can use to allocate, manipulate, and deallocate memory from external code modules.

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

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

## Memory Allocation

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

### Static

With static 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. This memory remains allocated while the application runs. This form of memory management 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

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 two kinds of dynamic memory allocation. The more conventional method uses pointers to allocate memory. With pointers, you request a block of memory of a certain size and the routine returns the address of the block to your CIN. When you no longer need the block of memory, you call a routine to free the block. You can use the block of memory to store data, and you reference that data using the address 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. This avoids problems that occur when you need to change the amount of memory allocated to a pointer, because other references would be out of date. If you need more memory, there might not be sufficient memory to expand the pointer's memory space without moving the memory block to a new location. This causes problems if an application had multiple references to the pointer, because each pointer refers to the old memory address of the data. Using invalid pointers can cause severe problems.

A second form of memory allocation uses handles to address this problem. As with pointers, when you allocate memory using handles, you request a block of memory of a 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 two distinct sections, called zones. LabVIEW uses the data space (DS) zone only to hold VI execution data. LabVIEW uses the application zone (AZ) to hold all other data. Most memory manager functions have two corresponding routines, one for each of the two zones. Routines that operate on the data space zone begin with `DS` and routines for the application zone begin with `AZ`.

Currently, the two zones are actually one zone, but this may change in future releases of LabVIEW; therefore, 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 one of these handles to a CIN, your CIN should use AZ routines to work with the handle.

## Using Pointers and Handles

Most memory manager functions have a DS routine and an AZ routine. In 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. Refer to the following section for a simple example of how to work with pointers and handles in C.

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

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

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

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

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

The last line frees the pointer.

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

```
int32 **myInt32H;

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

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

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

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

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

## Identifying Files and Directories

When you perform operations on files and directories, you need to identify the target of the operation. The platforms LabVIEW supports use a hierarchical file system, meaning files are stored in directories, possibly nested several levels deep. These file systems support the connection of multiple discrete storage media, called volumes. For example, DOS-based systems support multiple drives connected to the system. For most of these file systems, you must include the volume name to 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 target’s path. The path describes the volume containing the target, the directories between the top level and the target, and the target’s name. If the target is an open file, 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 three kinds of filepath specifications: conventional, empty, and 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, and the last name is the file or directory the path identifies.

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

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

For instance, in 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 |

In 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 |

In Macintosh, you separate names in a path with the colon (`:`) character. If the path is an absolute path, you begin the specification with the name of the volume containing the file. If an absolute path consists of only one name (it specifies a volume), it must end with a colon. If the path is a relative path, it begins with a colon. This colon is optional for a relative path consisting of only one name. 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.

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

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

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

### LabVIEW

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

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 (4-byte length field followed by data, generally stored in a handle), Pascal strings (1-byte length field followed by data), and C strings (data terminated by a null character).

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

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

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

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

# 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, as described by the `.c` file LabVIEW creates when you right-click the node on the block diagram and select **Create .c File**. 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/initialize global state information, you need to understand more of how LabVIEW manages data and CINs, as described in the following sections.

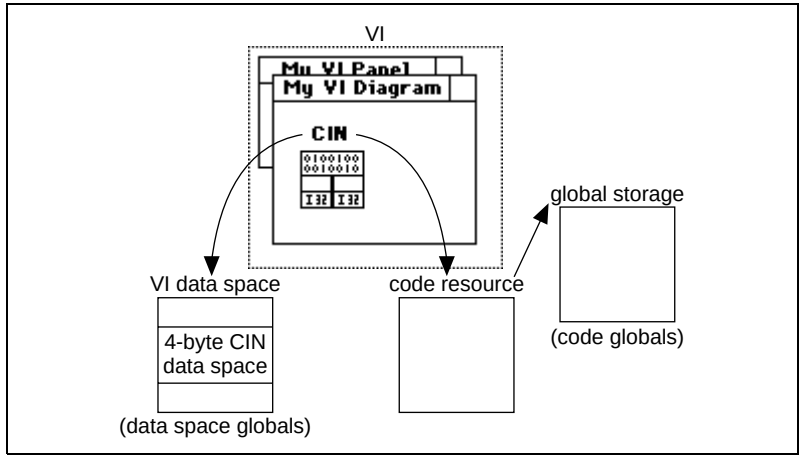
### Data Spaces and Code Resources

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

When LabVIEW loads a VI, it allocates a *data space*, a block of data storage memory, for that VI. LabVIEW uses this 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 *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 these globals 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 these global variables is ever in memory and their values are consistent.

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

This linking behaves the same way as links between VIs and subVIs. When you try to reference a subVI and another VI with the same name already exists in memory, LabVIEW references the one already in memory instead of the one you selected. In the same way, if you try to load references to two different code resources having the same name, only one code resource is

actually loaded into memory, and both references point to the same code. However, LabVIEW can verify that a subVI call matches the subVI connector pane terminal, but LabVIEW cannot verify that your source code matches the CIN call.

## 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 non-reentrant. The other cases have slightly more complicated behavior, described in later sections of this chapter.

### Loading a VI

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

LabVIEW calls `CINLoad` once for a given code resource, regardless of the number of data spaces and the number of references to that code resource. For this reason, 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 *Code Globals and CIN Data Space Globals*, *Loading a New Resource into the CIN*, and *Compiling a VI* sections later in this chapter for more information.

### Unloading a VI

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

When a VI is unloaded from memory, LabVIEW calls the `CINDispose` routine, giving you a chance to dispose of anything you allocated earlier. `CINDispose` is called for each `CINInit` call. For instance, if you used *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/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. The new code resource is then given a chance to perform any initialization it needs to perform. LabVIEW calls the CINLoad for the new code resource, followed by the CINInit routine, called once for each data space and each reference to the code resource.

## Compiling a VI

When you compile a VI, LabVIEW recreates the VI data space, including resetting all uninitialized shift registers to their default values. In the same way, your CIN is given a chance to dispose or initialize any storage it manages. Before disposing of the current data space, LabVIEW calls the CINDispose routine for each reference to the code resource within the VI(s) being compiled to give the code resource a chance to dispose of any old results it is managing. LabVIEW then compiles the VI and creates a new data space for the VI(s) being compiled (multiple data spaces for any reentrant VI). LabVIEW then calls CINInit for each reference to the code resource within the compiled VI(s) to give 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, giving 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, it is called for each CIN data space as well. CINs in VIs not currently running are not notified by LabVIEW of the abort event.

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

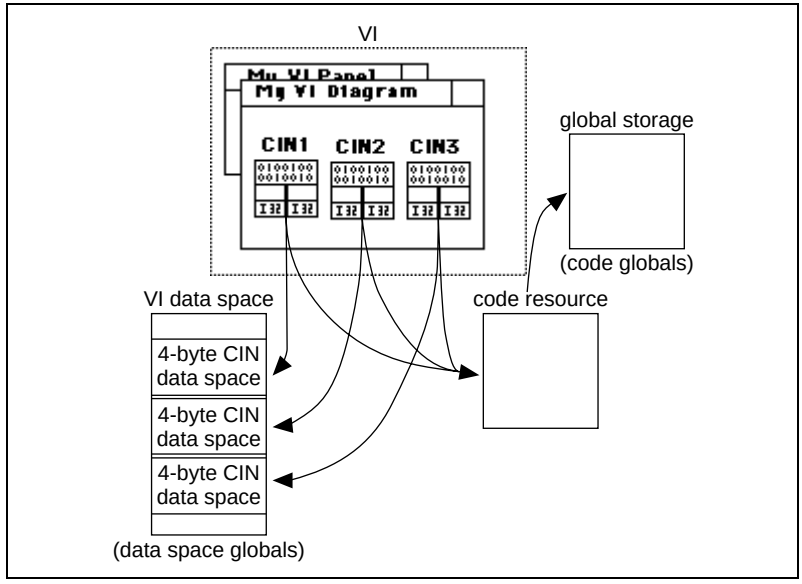
## Multiple References to the Same CIN in a Single VI

If you loaded the same code resource into multiple CINs, or 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 (though 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 its CIN data space might need initialization. Thus, if you have two nodes in the same VI, where 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. LabVIEW has already called CINLoad once, and does not call it 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.

The following illustration shows an example of 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. To use multithreading, you must use LabVIEW on Windows or Solaris 2.*x*.

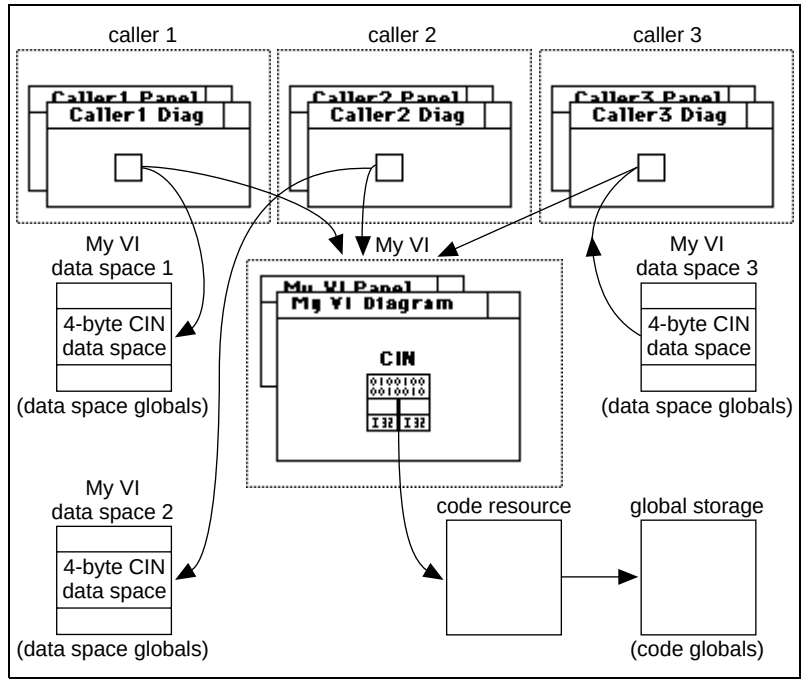
## Single-Threaded Operating Systems

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

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

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.

The following illustration shows an example of three VIs referencing a reentrant VI containing one CIN.




## Multithreaded Operating Systems

By default, CINs written before LabVIEW 5.0 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. If you want a CIN to run in the current execution thread of the block diagram, add the following code to your `.c` file:

```
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, but `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 it, 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 this storage location 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 is an example of the exact syntax of these two routines, defined in `extcode.h`.

- `int32 GetDSStorage(void);`

  This routine returns the value of the 4-byte quantity in the CIN data space LabVIEW allocates for each CIN code resource, or for each use

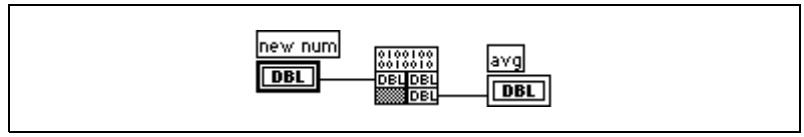
of the surrounding VI (if the VI is reentrant). Call this routine only from `CINInit`, `CINDispose`, `CINAbort`, or `CINRun`.

- `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`.

## 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.




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 nodes 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 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. The variables are initialized in `CINLoad`. If the variables are dynamically created (if they are pointers or handles), you can allocate the memory for the pointer or handle in `CINLoad` and deallocate it in `CINUnload`. You can do this because `CINLoad` and `CINUnload` are called only once, regardless of the number of references to the code resources and the number of data spaces. This example does not use the `UseDefaultCINLoad` macro, because this `.c` file has a `CINLoad` function.

```
/*
 * 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;
   }
```

## Using CIN Data Space Globals

The following code averages using CIN data space globals. A handle for the global data is allocated in `CINInit`, and stored in the CIN data space storage using `SetDSStorage`. When LabVIEW calls the `CINInit`, `CINDispose`, `CINAbort`, or `CINRun` routines, it 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 that data, use `GetDSStorage` to retrieve the handle and then dereference the appropriate fields. Finally, use the `CINDispose` routine you need to dispose of the handle.

```
/*
 * 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;
    }
```

# 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 in 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.

To perform the following operations, use the functions listed:

- Handle and pointer verification:
  - `AZCheckHandle/DSCheckHandle`
  - `AZCheckPtr/DSCheckPtr`
- Handles, allocating and releasing:
  - `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:
  - AZHandAndHand/DSHandAndHand
  - AZHandToHand/DSHandToHand
  - AZPtrAndHand/DSPtrAndHand
  - AZPtrToHand/DSPtrToHand
  - AZPtrToXHand/DSPtrToXHand
  - ClearMem
  - MoveBlock
  - SwapBlock
- Memory zone utilities:
  - AZHeapCheck/DSHeapCheck
  - AZMaxMem/DSMaxMem
  - AZMemStats/DSMemStats
- Pointers, allocating and releasing:
  - AZDisposePtr/DSDisposePtr
  - AZNewPClr/DSNewPClr
  - AZNewPtr/DSNewPtr

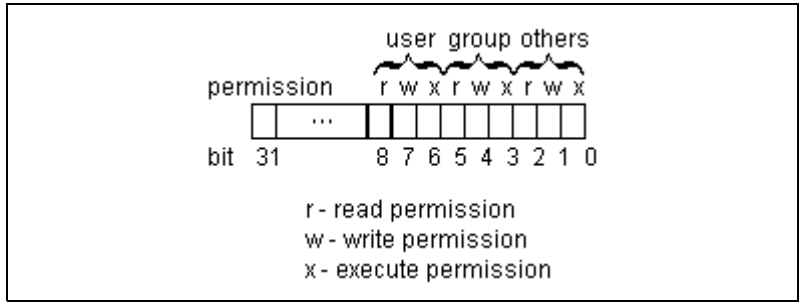
# File Manager Functions

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.

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`.

In UNIX, the nine bits of permissions correspond exactly to nine UNIX permission bits governing read, write, and execute permissions for user, group, and others. The following illustration shows permission bits in UNIX.




In 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.

In Macintosh, 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 Macintosh.

To perform the following operations, use 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
  - FlsARefNum
  - 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
  - FlsAPathOfType
  - FSetPathType
- Path, extracting information:
  - FDepth
  - FDirName
  - FName
  - FNamePtr
  - FVolName

- Paths, comparing:
  - FlsAPath
  - FlsAPathOrNotAPath
  - FlsEmptyPath
  - 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:
  - FDisposePath
- 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.

To perform the following operations, use 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
- Utility functions:
  - BinSearch
  - QSort
  - Unused
- Time functions:
  - ASCIITime
  - DateCString
  - DateToSecs
  - MilliSecs

- SecsToDate
- TimeCString
- TimeInSecs

## Mathematical Operations

In addition to the mathematical operations in the previous 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);
```

# 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 AM, GMT. This function uses the same date format as that returned by the `DateCString` function using a mode of `2`. The date is followed by a space, and the time is in the same format as that returned by the `TimeCString` function using a mode of `0`. 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 AM, GMT. |

## 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 *XX*`NewPtr` or *XX*`NewPClr`. 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. |

## 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.

# AZHandAndHand/DSHandAndHand

```
MgErr AZHandAndHand(h1, h2);
MgErr DSHandAndHand(h1, h2);
```

## Purpose

Appends the data referenced by **h1** to the end of the memory block referenced by **h2**.

The function resizes handle **h2** to hold **h1** and **h2** data. If **h1** is an AZ handle, lock it, because this routine can move memory.

## Parameters

| Name | Type | Description |
|---|---|---|
| **h1** | `UHandle` | Source of data you want to append to **h2**. |
| **h2** | `UHandle` | Initial handle, to which the data of **h1** is appended. |

## 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. |

# AZHandToHand/DSHandToHand

```
MgErr AZHandToHand(hp);
MgErr DSHandToHand(hp);
```

## Purpose

Copies the data referenced by the handle to which **hp** points into a new handle, and returns a pointer to the new handle in **hp**.

Use this routine to copy an existing handle into a new handle. The old handle remains allocated. This routine writes over the pointer that is passed in, so you should maintain a copy of the original handle.

## Parameters

| Name | Type | Description |
|---|---|---|
| **hp** | `UHandle` | Pointer to handle you want to duplicate. A pointer to the resulting handle is returned in this parameter. Refer to the *Pointers as Parameters* section in Chapter 3, *CINs*, for more information about using this parameter. |

## 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. |

# 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; it 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 in Chapter 3, *CINs*, for more information about using this parameter. |

A `MemStatRec` structure is defined as follows:

```
typedef struct {
   int32 totFreeSize, maxFreeSize, nFreeBlocks;
   int32 totAllocSize, maxAllocSize;
   int32 nPointers, nUnlockedHdls, nLockedHdls;
   int32 reserved [4];
}
```

The free memory in a zone consists of a number of blocks of contiguous memory. In the `MemStatRec` structure, **totFreeSize** is the sum of the sizes of these blocks, **maxFreeSize** is the largest of these blocks (as returned by *XX*`MaxMem`), and **nFreeBlocks** is the number of these blocks.

Similarly, the allocated memory in a zone consists of a number of blocks of contiguous memory. In the `MemStatRec` structure, **totAllocSize** is the sum of the sizes of these blocks and **maxAllocSize** is the largest of these blocks.

Because there are three different varieties of allocated blocks, the numbers of blocks of each type is returned separately.

**nPointers** (`int32`) is the number of pointers, **nUnlockedHdls** (`int32`) is the number of unlocked handles, and **nLockedHdls** (`int32`) is the number of locked handles. Add these three values together to find the total number of allocated blocks.

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

# 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`.

# AZPtrAndHand/DSPtrAndHand

```
MgErr AZPtrAndHand(p, h, size);
MgErr DSPtrAndHand(p, h, size);
```

## Purpose

Appends **size** bytes from the address referenced by **p** to the end of the memory block referenced by **h**.

## Parameters

| Name | Type | Description |
|---|---|---|
| **p** | `UPtr` | Source of data you want to append to **h**. |
| **h** | `UHandle` | Handle to which the data of **p** is appended. |
| **size** | `int32` | Number of bytes to copy from **p**. |

## 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. |

# AZPtrToHand/DSPtrToHand

```
MgErr AZPtrToHand(p, hp, size);
MgErr DSPtrToHand(p, hp, size);
```

## Purpose

Creates a new handle of **size** bytes and copies **size** bytes from the address referenced by **p** to the handle.

## Parameters

| Name | Type | Description |
|---|---|---|
| **p** | `UPtr` | Source of data you want to copy to the handle pointed to by **hp**. |
| **hp** | `UHandle` | Pointer to handle you want to duplicate. A pointer to the resulting handle is returned in this parameter. Refer to the *Pointers as Parameters* section in Chapter 3, *CINs*, for more information about using this parameter. |
| **size** | `int32` | Number of bytes to copy from **p** to the new handle. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `NoErr` | No error. |
| `MFullErr` | Not enough memory to perform the operation. |

# AZPtrToXHand/DSPtrToXHand

```
MgErr AZPtrToXHand(p, h, size);
MgErr DSPtrToXHand(p, h, size);
```

## Purpose

Copies **size** bytes from the address referenced by **p** to the existing handle **h**, resizing **h**, if necessary, to hold the results.

## Parameters

| Name | Type | Description |
|---|---|---|
| **p** | `UPtr` | Source of data you want to copy to the handle **h**. |
| **h** | `UHandle` | Destination handle. |
| **size** | `int32` | Number of bytes to copy from **p** to the existing 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. |

# 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`, described in the *Resizing Arrays and Strings* section in Chapter 4, *Programming Issues for CINs*. 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 `x` 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, and 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 errors:

`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:

`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 ***t*** seconds after January 1, 1904, 12:00 AM, GMT. 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 AM, GMT. |
| **fmt** | `int32` | Indicates the format of the returned date string, using the following values:<br>• `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`.<br>• `1`—Long date format, *`dayName`*`, `*`MonthName`*`, `*`DayOfMonth`*`, `*`LongYear`*. For example, `Thursday, December 31, 1992`.<br>• `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 AM, GMT.

## 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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

The corresponding number of seconds since January 1, 1904, 12:00 AM, GMT.

# FAddPath

```
MgErr FAddPath(basePath, relPath, newPath);FAddPath
```

## 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:

- `FAddPath(basePath, relPath, newPath);`
  `/* the new path is returned in a third path variable */`
- `FAddPath(path, relPath, path);`
  `/* the new path writes over the old base path */`
- `FAddPath(basepath, path, path);`
  `/* the new path writes over the old relative path */`

## 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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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 Macintosh, **group** is ignored.

If the function creates the file, the resulting file descriptor is stored in the address referred to by **fdp**. If an error occurs, the function stores `0` in the address referred to by **fdp** and returns an error.

**Note** Before you call this function, make sure that you understand how to use the **fdp** parameter. Refer to the *Pointers as Parameters* section in Chapter 3, *CINs*, for more information about using this 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 in 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`:<br>• `openReadOnly`—Open for reading.<br>• `openWriteOnly`—Open for writing.<br>• `openReadWrite`—Open for both reading and writing. |

| Name | Type | Description |
|---|---|---|
| **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`:<br>• `denyReadWrite`—Prevents others from reading from and writing to the file while it is open.<br>• `denyWriteOnly`—Prevents others from writing to the file only while it is open.<br>• `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:

| | |
|---|---|
| `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 Macintosh 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 Macintosh, **group** is ignored.

If the function creates the file, the resulting file descriptor is stored in the address referred to by **fdp**. If an error occurs, the function stores `0` in the address referred to by **fdp** and returns an error.

**Note** Before you call this function, make sure that you understand how to use the **fdp** parameter. Refer to the *Pointers as Parameters* section in Chapter 3, *CINs*, for more information about using this 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 in 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`:<br>• `openReadOnly`—Open for reading.<br>• `openWriteOnly`—Open for writing.<br>• `openReadWrite`—Open for both reading and writing. |

| Name | Type | Description |
|---|---|---|
| **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`:<br>• `denyReadWrite`—Prevents others from reading from and writing to the file while it is open.<br>• `denyWriteOnly`—Prevents others from writing to the file only while it is open.<br>• `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:

| | |
|---|---|
| `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 Macintosh 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 (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. |

# 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:

`mgArgErr` A bad argument was passed to the function. Verify the path.

# FDisposePath

```
MgErr FDisposePath(p);
```

## Purpose

Disposes of a path.

## Parameters

| Name | Type | Description |
|---|---|---|
| **p** | `Path` | Path you want to dispose of. |

## Return Value

`mgErr`, which can contain the following errors:

`mZoneErr` Invalid 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 errors:

`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 a 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 in 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:

| | |
|---|---|
| `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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `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 in 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:

| | |
|---|---|
| `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:<br>• `fAbsPath`—The path is absolute.<br>• `fRelPath`—The path is relative.<br>• `fNotAPath`—The path is the canonical invalid path or an error occurred.<br>Refer to the *Pointers as Parameters* section in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

`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 in 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:

| | |
|---|---|
| `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 Macintosh 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

This function is similar to `FileNameCmp`, except you pass the function handles to the string data instead of pointers. Use this function to compare two file names lexically and 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 Macintosh and Windows, case-sensitive for SPARCstation.

## 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 Macintosh 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:<br>• `fAbsPath`—Compare the path's type to absolute.<br>• `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

```
Bool32FIsARefNum(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 *Basic Data Types* section in 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 file type record is:

```
typedef struct {
   int32 flags;
   int32 type;
} FileType;
```

Only the least significant four bits of flags contain useful information. The remaining bits are reserved for use by LabVIEW. You can test these four bits using the following four masks:

```
#define kIsFile 0x01
#define kRecognizedType 0x02
```

```
#define kIsLink 0x04
#define kFIsInvisible 0x08
```

The kIsFile bit is set if the item described by the file type record is a file; otherwise, it is clear. The kRecognizedType bit is set if the item described is a file for which you can determine a 4-character file type; otherwise, it is clear. The kIsLink bit is set if the item described is a UNIX link or Macintosh alias; otherwise, it is clear. The kFIsInvisible bit is set if the item described does not appear in a file dialog; otherwise, it is clear.

The value of type is defined only if the kRecognizedType bit is set in flags. In this case, type is the 4-character file type of the file described by the file type record. This 4-character file type is provided by the file system in Macintosh and is computed by examining the file name extension on other systems.

## Return Value

mgErr, which can contain the following errors:

| | |
|---|---|
| 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:<br>• `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`).<br>• `fCurrent`—The first byte to lock or unlock is located **offset** bytes from the current position mark (**offset** can be positive, `0`, or negative).<br>• `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 errors:

| | |
|---|---|
| `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 `FDisposePath`.

## 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 Macintosh 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:

| | |
|---|---|
| `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 in Chapter 3, *CINs*, for more information about using this 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 in 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`:<br>• `openReadOnly`—Open for reading.<br>• `openWriteOnly`—Open for writing; file is not truncated (data is not removed). In Macintosh, 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.<br>• `openReadWrite`—Open for both reading and writing.<br>• `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`:<br>• `denyReadWrite`—Prevents others from reading from and writing to the file while it is open.<br>• `denyWriteOnly`—Prevents others from writing to the file only while it is open.<br>• `denyNeither`—Allows others to read from and write to the file while it is open. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `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 Macintosh 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:

| | |
|---|---|
| `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 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 in 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:

| | |
|---|---|
| `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:<br>• `fStart`—Current position mark moves to **ofst** bytes relative to the start of the file (**ofst** must be greater than or equal to `0`).<br>• `fCurrent`—Current position mark moves **ofst** bytes from the current position mark (**ofst** can be positive, `0`, or negative).<br>• `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:

| | |
|---|---|
| `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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `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 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 in 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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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, so **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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `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** was 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:

`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 in 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:

| | |
|---|---|
| `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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

| | |
|---|---|
| `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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

`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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

`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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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:

| | |
|---|---|
| `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:<br>• `fAbsPath`—The path is absolute.<br>• `fRelPath`—The path is relative. |

## Return Value

`mgErr`, which can contain the following errors:

`mgArgErr` Badly formed path or invalid type.

# FStrFitsPat

```
Bool32FStrFitsPat(pat, str, pLen, sLen);
```

## Purpose

Determines whether a filename, **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.<br>\ is literal, not treated as having a special meaning. A single backslash at the end of **pat** is the same as two backslashes.<br>? matches any one character.<br>* matches zero or more characters. |
| **str** | `uChar *` | Filename (string) to compare to pattern. |
| **pLen** | `int32` | Number of characters in **pat**. |
| **sLen** | `int32` | Number of characters in **str**. |

## Return Value

`Bool32`, which can contain the following values:

| | |
|---|---|
| `TRUE` | Filename fits the pattern. |
| `FALSE` | Filename 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 non-zero, 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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

`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 non-zero, 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 in Chapter 3, *CINs*, for more information about using this parameter. |

## Return Value

`mgErr`, which can contain the following errors:

`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 in 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 errors:

`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**, $0 \leq \mathbf{n} \leq 15$.

## 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. |

# 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.

# PPStrCaseCmp

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

## Purpose

This function is similar to `PStrCaseCmp`, except you pass the function handles to the string data instead of pointers. Use this function to compare two Pascal strings lexically and determine whether one is less than, equal to, or greater than the other. This comparison ignores differences in case.

## 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

This function is similar to `PStrCmp`, except you pass the function handles to the string data instead of pointers. Use this function to compare two Pascal strings lexically and determine whether one is less than, equal to, or greater than the other. This comparison is case sensitive.

## 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, and each argument follows the format string paired with a conversion specification embedded in the format string. The second parameter, **cfmt** or **pfmt**, must be cast appropriately to either type `CStr` or `PStr`.

`SPrintf` prints to a C string, just like the C library function `sprintf`. `sprintf` returns the actual character count and appends a `NULL` byte to the end of the destination C string.

`SPrintfp` is the same as `SPrintf`, except the format string is a Pascal string instead of a C string. As with `SPrintf`, `SPrintfp` appends a `NULL` byte to the end of the destination C string.

If you pass `NULL` for **destCStr**, `SPrintf` and `SPrintfp` do not write data to memory, and they return the number of characters required to contain the resulting data, not including the terminating `NULL` character.

`PPrintf` prints to a Pascal string with a maximum of 255 characters. `PPrintf` sets the length byte of the Pascal string to reflect the size of the resulting string. `PPrintf` does not append a `NULL` byte to the end of the string.

`PPrintfp` is the same as `PPrintf`, except the format string is a Pascal string instead of a C string. As with `PPrintf`, `PPrintfp` sets the length byte of the Pascal string to reflect the size of the resulting string.

`FPrintf` prints to a file specified by the refnum in **fd**. `FPrintf` does not embed a length count or a terminating `NULL` character in the data written to the file.

`LStrPrintf` prints to a LabVIEW string specified by **destLsh**. Because the string is a handle that may be resized, `LStrPrintf` can return memory errors just as `DSSetHandleSize` does.

These functions accept the following special characters:

| | |
|---|---|
| `\b` | Backspace |
| `\f` | Form feed |

| | |
|---|---|
| `\n` | New line (inserts the system-dependent end-of-line char(s); for example, CR on Macintosh, NL on UNIX, CRNL on DOS) |
| `\r` | Carriage return |
| `\s` | Space |
| `\t` | Tab |
| `%%` | Percentage character (to print %) |

These functions accept the following formats:

`%[-] [field size] [.precision] [argument size] [conversion]`

| | |
|---|---|
| `[-]` | 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 word (16-bit integer). The flag l causes this conversion to convert the data so the function assumes the data is a long 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 Specifier | Description |
|---|---|
| b | Binary |
| c | Print a character (`%2c, %4c` print on `int16`, `int32` as a `2,4 char` constant) |
| d | Decimal |
| e | Exponential |

| Conversion Specifier | Description |
|---|---|
| f | Fixed-point format |
| H | String handle (LStrHandle) |
| o | Octal |
| p | Pascal string |
| P | Long Pascal string (LStrPtr) |
| q | Print a point (passed by value) as %d,%d representing horizontal, vertical coordinates |
| Q | Print a point (passed by value) as hv(%d,%d) representing horizontal, vertical coordinates |
| r | Print a rectangle (passed by reference) as %d,%d,%d,%d representing top, left, bottom, right coordinates |
| R | Print a rectangle (passed by reference) as tlbr(%d,%d,%d,%d) representing top, left, bottom, right coordinates |
| s | String |
| u | Unsigned decimal |
| x | Hex |
| z | Path |

# 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 in 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 AM, GMT into a data structure containing numerical information about the date, including the year (`1904` through `2040`), the month (`1` through `12`), the day as it corresponds to the current year (`1` through `366`), month (`1` through `31`), and week (`1` through `31`), hour (`0` through `23`), the hour (`0` through `23`), minute (`0` through `59`), and second (`0` through `59`) of that day, and a value indicating whether the time specified uses daylight savings time.

## Parameters

| Name | Type | Description |
|---|---|---|
| **secs** | `uInt32` | Seconds since January 1, 1904, 12:00 AM, GMT. |
| **dateRecordP** | `DateRec *` | Pointer to a `DateRec` structure. `SecsToDate` stores the converted date in the fields of the date structure referred to by **dateRecordP**. Refer to the *Pointers as Parameters* section in 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 in 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 AM, GMT. 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 AM, GMT. |
| **fmt** | `int32` | Indicates the format of the returned time string, using the following values:<br>• `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`).<br>• `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 AM, GMT.

# 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

# Common Questions

**What languages can I use to write DLLs?**

Any language can be used to write DLLs as long as the DLL can be called using one of the calling conventions that LabVIEW supports (stdcall or C).

**Why is it no longer possible to build external subroutines in LabVIEW?**

External subroutines provided a solution for users who wanted to share code among multiple CINs. At the time that LabVIEW first provided for external subroutines, shared libraries (DLLs) were not yet commonplace. Since shared libraries are now widely used, and since they provide all the functionality that external subroutines did, National Instruments decided to drop support for the creation of external subroutines. Users who want to share code among multiple CINs should use shared libraries.

**Why does the "Function Name" ring contain an empty list of functions for certain DLLs?**

On Windows platforms, the most likely reason is that the DLL is 16-bit. LabVIEW cannot call 16-bit DLLs. It is also possible, though unlikely, that the DLL has no exported functions. The UNIX platforms do not implement this functionality.

**Why does the function I wish to call not appear in the "Function Name" ring of the Call Library Function configuration dialog?**

The most likely reason is that the function has not been exported. See the documentation for your compiler for information about how to mark functions for export.

**Why does LabVIEW crash when I call a function in my DLL?**

The most likely causes are: 1) an error in the calling convention you have specified in the **Call Library Function** configuration dialog; 2) one of the function parameters being of incorrect type; and 3) an error in the code of the DLL, such as dereferencing a null pointer.

**In the Function Prototype section of the Call Library Function configuration dialog, why does the function name have unusual characters appended?**

The function name that appears in the function prototype section will have characters such as "@" appended if the function was "decorated" when the DLL was built. This is most common with C++ compilers. This is normal and not a cause for concern. The undecorated name will appear in the Function Name ring of the configuration dialog.

**Why do I receive memory.cpp errors when I call a function in my DLL?**

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. Note that these kinds of crashes may or may not occur at the time the DLL call actually executes on the block diagram.

**Is it possible to return a pointer from a call to a function in a DLL?**

Strictly speaking, this is not possible, because there are no pointer types in LabVIEW. However, you can specify the return type to be an integer that is the same size as the pointer. LabVIEW will then treat the address as a simple integer.

**Is it possible to allocate memory using malloc inside a CIN?**

Yes, but the pointer that results from the malloc call should be assigned to a variable that is local to the CIN code, rather than to a variable passed from the LabVIEW diagram. You should use LabVIEW memory manager functions if you wish to create or resize memory associated with a variable passed from the LabVIEW diagram.

**Can CINs be written in a language other than C?**

This is technically possible if the CIN entry points (i.e. CINRun, CINLoad, etc.) are declared as `extern "C"`. However, National Instruments recommends using a DLL rather than a CIN if you wish to use a language other than C or C++.

**What are the advantages of using a CIN rather than a DLL?**

The advantages are: 1) the CIN code is integrated into the code of the VI, so there is no extra file to maintain when the VI is distributed; 2) CINs provide certain special entry points (CINLoad, CINSave, etc.).

**What are the advantages of using a DLL rather than a CIN?**

The advantages are: 1) you can change the DLL without changing any of the VIs that link to the DLL (provided you do not modify the function prototypes); 2) Practically all modern development environments provide excellent support for creating DLLs, while LabVIEW supports only a subset of development environments for creating CINs.

**Is it possible to call the LabVIEW manager functions from a DLL?**

Yes. You need to #include extcode.h in any files that use manager functions, and you must link to `labview.lib`. You should also set your compiler's structure alignment to 1 byte. Note that some of the manager functions, such as `SetCINArraySize`, are CIN-specific and may not be called from a DLL.

**Is it faster to call a DLL or a CIN, assuming the underlying code is the same?**

There is no difference in speed.

**One or more of the parameters of the function I wish to call in a DLL are of types that do not exist in LabVIEW. Can I still call this function from LabVIEW?**

You can call the function, but you must ensure that each parameter is passed to the function in a way that allows the DLL to correctly interpret the data. Starting in LabVIEW 6.0, the Call Library Function allows you to create a skeleton `.c` file from its current configuration. By viewing this C file, you can determine whether LabVIEW will pass the data in a manner compatible with the DLL function, and make necessary adjustments.

# B

# Technical Support Resources

## Web Support

National Instruments Web support is your first stop for help in solving installation, configuration, and application problems and questions. Online problem-solving and diagnostic resources include frequently asked questions, knowledge bases, product-specific troubleshooting wizards, manuals, drivers, software updates, and more. Web support is available through the Technical Support section of `www.ni.com`

## NI Developer Zone

The NI Developer Zone at `zone.ni.com` is the essential resource for building measurement and automation systems. At the NI Developer Zone, you can easily access the latest example programs, system configurators, tutorials, technical news, as well as a community of developers ready to share their own techniques.

## Customer Education

National Instruments provides a number of alternatives to satisfy your training needs, from self-paced tutorials, videos, and interactive CDs to instructor-led hands-on courses at locations around the world. Visit the Customer Education section of `www.ni.com` for online course schedules, syllabi, training centers, and class registration.

## System Integration

If you have time constraints, limited in-house technical resources, or other dilemmas, you may prefer to employ consulting or system integration services. You can rely on the expertise available through our worldwide network of Alliance Program members. To find out more about our Alliance system integration solutions, visit the System Integration section of `www.ni.com`

# Worldwide Support

National Instruments has offices located around the world to help address your support needs. You can access our branch office Web sites from the Worldwide Offices section of `www.ni.com`. Branch office web sites provide up-to-date contact information, support phone numbers, e-mail addresses, and current events.

If you have searched the technical support resources on our Web site and still cannot find the answers you need, contact your local office or National Instruments corporate. Phone numbers for our worldwide offices are listed at the front of this manual.

# Glossary

## A

| | |
|---|---|
| ANSI | American National Standards Institute. |
| application zone | *See* AZ. |
| 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* object code. *See* 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 options pertain to that object specifically. |
| source code | Original, uncompiled text code. |
| source terminal | Terminal that emits data. |

## T

| | |
|---|---|
| type descriptor | *See* data type descriptor. |