File Manager Functions

This topic contains descriptions of the file manager functions that perform the following operations:

Current Position Mark, Positioning
Default Access Rights Information, Getting
Directory Contents, Creating and Determining
End-Of-File Mark, Positioning
File Data to Disk, Flushing
File, Directory, and Volume Information Determination
File Operations, Performing Basic
File Range, Locking
File Refnums, Manipulating
Filenames and Patterns, Matching
Files and Directories, Moving and Deleting
Files, Copying
Paths, Comparing
Paths, Converting to and from Other Representations
Paths, Creating
Paths, Disposing
Paths, Duplicating
Path, Extracting Information
Path Type, Determining

Current Position Mark, Positioning

FMSeek
FMTell

Click here to view a list of all File Manager Functions.

FMSeek

syntax `MgErr FMSeek(fd, ofst, mode);`

`FMSeek` sets the current position mark for a file to the specified point, relative to the beginning of the file, the current position in the file, or the end of the file. If an error occurs, the current position mark does not move.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file.
ofst	`int32`	New position of the current position mark. The position is the number of bytes from the beginning of the file, the current position mark, or the end of the file, as determined by **mode**.
mode	`int32`	Position in the file relative to which `FMSeek` sets the current position mark for a file. If **mode** is `fStart`, the current position mark moves to **ofst** bytes relative to the start of the file (**ofst** must be greater than or equal to 0). If **mode** is `fCurrent`, the current position mark moves **ofst** bytes from the current position mark (**ofst** can be positive, 0, or negative). If **mode** is `fEnd`, the current position mark moves to **ofst** bytes from the end of the file (**ofst** must be less than or equal to 0).

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

Error	Description
`mgArgErr`	Not a valid file descriptor.
`fEOF`	Attempt to seek before the start or after the end of the file.
`fIOErr`	Unspecified I/O error occurred.

FMTell

syntax `MgErr FMTell(fd, ofstp);`

`FMTell` **returns** the position of the current position mark in the file.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file.
ofstp	`int32 *`	Address at which `FMTell` stores the position of the current position mark, in terms of bytes relative to the

beginning of the file. If an error occurs, the contents of **ofstp** is undefined.

See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	**Description**
mgArgErr	Not a valid file descriptor.
fIOErr	Unspecified I/O error occurred.

Default Access Rights Information, Getting

FGetDefGroup

Click here to view a list of all File Manager Functions.

FGetDefGroup

syntax `LStrHandle FGetDefGroup(groupHandle);`

`FGetDefGroup` gets the LabVIEW default group for a file or directory.

Parameter	Type	Description
groupHandle	`LStrHandle`	Handle that represents the LabVIEW default group for a file or directory. If **groupHandle** is `NULL`, `FGetDefGroup` allocates a new handle and **returns** the default group in it. If **groupHandle** is a handle, `FGetDefGroup` **returns** it, and **groupHandle** resizes to hold the default group.

returns The resulting `LStrHandle`; if **groupHandle** was not `NULL`, then the return value is the same `LStrHandle` as **groupHandle**. If an error occurs, NULL is returned.

Directory Contents, Creating and Determining

FListDir
FNewDir

Click here to view a list of all File Manager Functions.

FListDir

syntax `MgErr` `FListDir(path, list, typeH);`

`FListDir` determines the contents of a directory.

The function fills the (`AZ`) handle passed in **list** with a `CPStr`, where the **cnt** field specifies the number of concatenated Pascal strings that follow in the `str`[] field. See the *Dynamic Data Types* section of Chapter 5, *Manager Overview*, in the *Code Interface Reference Manual* for a description of the `CPStr` data type. If **typeH** is not `NULL`, the function fills the AZ handle passed in **typeH** with the file type information for each file name or directory name stored in **list**.

Parameter	Type	Description
path	`Path`	Path of the directory whose contents you want to determine.
list	`CPStrHandle`	Application zone handle in which `FListDir` stores a series of concatenated Pascal strings, preceded with a 4-byte integer field, **cnt**, that indicates the number of items in the buffer.
typeH	`FileType`	Application zone handle in which `FListDir` stores a series of `FileType` records. If **typeH** is not `NULL`, then `FListDir` stores one `FileType` record in **typeH** for each Pascal string in list. The *n*th `FileType` in **typeH** denotes the file type information about the file or directory named in the *n*th string in **list**.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fNotFound`	Directory not found.
`fNoPerm`	Access denied (file/directory/disk is locked/protected).
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error occurred.

FNewDir

syntax `MgErr` `FNewDir(path, permissions);`

`FNewDir` creates a new directory with the specified **permissions**. If an error occurs, the function does not create the directory.

Parameter	Type	Description
path	`Path`	Path of the directory you want to create.

permissions	`int32`	Permissions for the new directory. For a description of permissions, see the topic file permissions.

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

Error	**escription**
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fNoPerm`	Access denied (file/directory/disk is locked /protected).
`fDupPath`	Directory already exists.
`fIOErr`	Unspecified I/O error occurred.

End-Of-File Mark, Positioning

FGetEOF
FSetEOF

Click here to view a list of all File Manager Functions.

FGetEOF

syntax `MgErr` `FGetEOF(fd, sizep);`

`FGetEOF` **returns** the size of the specified file.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file.
sizep	`int32 *`	Address at which `FGetEOF` stores the size of the file in bytes. If an error occurs, the contents of ***sizep** is undefined. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	Description
`mgArgErr`	Not a valid file descriptor.
`fIOErr`	Unspecified I/O error occurred.

FSetEOF

syntax `MgErr` `FSetEOF(fd, size);`

`FSetEOF` sets the size of the specified file. If an error occurs, the file size does not change.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file.
size	`int32`	New file size in bytes.

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

Error	Description
`mgArgErr`	Not a valid file descriptor or **size** < 0.
`fDiskFull`	Disk is full.
`fNoPerm`	Access denied (file exists or something is locked/protected).
`fIOErr`	Unspecified I/O error occurred.

File Data to Disk, Flushing

FFlush

Click here to view a list of all File Manager Functions.

FFlush

Flushing File Data to Disk

syntax `MgErr FFlush(fd);`

`FFlush` writes any buffered data for the specified file out to the disk.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file.

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

Error	Description
`mgArgErr`	Not a valid file descriptor.
`fIOErr`	Unspecified I/O error occurred.

File, Directory, and Volume Information Determination

Click here to view a list of all File Manager Functions.

FExists

syntax `int32` `FExists(path);`

`FExists` **returns** information about the specified file or directory. It **returns** less information than `FGetInfo`, but it is much quicker on many platforms.

Parameter	Type	Description
path	`Path`	Path of the file or directory about which you want information.

returns `int32`, which is one of the following values.

Error	Description
`kFIsFile`	Specified item is a file.
`kFIsFolder`	Specified item is a directory or folder.
`kFNotExist`	Specified item does not exist.

FGetAccessRights

syntax `MgErr` `FGetAccessRights(path, owner, group, permPtr);`

`FGetAccessRights` **returns** access rights information about the specified file or directory.

Parameter	Type	Description
path	`Path`	Path of the file or directory about which you want access rights information.
owner	`PStr`	Address at which `FGetAccessRights` stores the owner of the file or directory.
group	`PStr`	Address at which `FGetAccessRights` stores the group of the file or directory.
permPtr	`int32 *`	Address at which `FGetAccessRights` stores the permissions of the file or directory. For a description of permissions, see the topic file permissions. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNotFound	File not found.
fIOErr	Unspecified I/O error occurred.

FGetInfo

syntax `MgErr FGetInfo(path, infop);`

`FGetInfo` **returns** information about the specified file or directory.

Parameter	Type	Description
path	`Path`	Path of the file or directory about which you want information.
infop	`FInfoPtr`	Address where `FGetInfo` stores information about the file or directory. If an error occurs, the information is undefined. See also the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Referenc Manual* for more information about using this parameter.

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

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNotFound	File not found.
fIOErr	Unspecified I/O error occurred.

FGetVolInfo

syntax `MgErr FGetVolInfo(path, vinfo);`

`FGetVolInfo` gets a path specification and information for the volume containing the specified file or directory.

Parameter	Type	Description
path	`Path`	Path of a file or directory contained on the volume from which you want to get information. This path is overwritten with a path specifying the volume containing the specified file or directory. If an error occurs, this path is undefined.
vinfo	`VInfoRec *`	Address at which `FGetVolInfo` stores the information about the volume. If an error occurs, the information is undefined. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fIOErr	Unspecified I/O error occurred.

FSetAccessRights

syntax MgErr FSetAccessRights(path, owner, group, permPtr);

FSetAccessRights sets access rights information for the specified file or directory. If an error occurs, no information changes.

Parameter	Type	Description
path	Path	Path of the file or directory for which you want to set access rights information.
owner	PStr	New owner that FSetAccessRights sets for the file or directory if **owner** is not NULL.
group	PStr	New group that FSetAccessRights sets for the file or directory if **group** is not NULL.
permPtr	int32 *	Address of new permissions that FSetAccessRights sets for the file or directory if **permPtr** is not NULL.

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

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNotFound	File not found.
fIOErr	Unspecified I/O error occurred.

FSetInfo

syntax MgErr FSetInfo(path, infop);

FSetInfo sets information for the specified file or directory. If an error occurs, no information changes.

Parameter	Type	Description
path	Path	Path of the file or directory for which you want to set information.
infop	FInfoPtr	Address of information FSetInfo sets for the file or directory.

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

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
fNotFound	File not found.
fIOErr	Unspecified I/O error occurred.

File Operations, Performing Basic

FCreate
FCreateAlways
FMClose
FMOpen
FMRead
FMWrite

Click here to view a list of all File Manager Functions.

FCreate

syntax `MgErr` `FCreate(fdp, path, permissions, openMode, denyMode, group);`

`FCreate` creates a file with the name and location specified by **path** and with the specified **permissions**, and opens it for writing and reading, as specified by **openMode**. If the file already exists, an error is returned.

You can use **denyMode** to control concurrent access to the file from within LabVIEW. The **group** parameter allows you to assign the file to a UNIX group; under Windows or Macintosh, **group** is ignored.

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

Note: Before attempting to call this function, make sure that you understand how to use the fdp parameter. See the *Pointers as Parameters* **section of** *Chapter 1, CIN Overview* **in the** *Code Interface Reference Manual* **for more information about this parameter.**

Parameter	Type	Description
fdp	`File *`	Address at which FCreate stores the file descriptor for the new file. If FCreate fails, it stores 0 in the address **fdp**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.
path	`Path`	Path of the file that you want to create.
permissions	`int32`	Permissions to assign to the new file. For a description of permissions, see the topic file permissions.
openMode	`int32`	Access mode to use in opening the file. Can have the following values, which are defined in the file `extcode.h`. • `openReadOnly`: Open for reading. • `openWriteOnly`: Open for writing • `openReadWrite`: Open for both reading and writing
denyMode	`int32`	Mode that determines what level of concurrent access to the file is allowed. Can have the following values, which are defined in the file `extcode.h`. • `denyReadWrite`: Prevents others from reading from and writing to the file while it is open.

		• `denyWriteOnly`: Prevents others from writing to the file only while it is open • `denyNeither`: allows others to read from and write to the file while it is open.
group	`PStr`	UNIX group you want to assign to the new file.

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

Error	**Description**
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fIsOpen`	File is already open for writing. This error is returned only on the Macintosh and the Sun. The PC **returns** `fIOErr` when the file is already open for writing.
`fNoPerm`	Access denied (something is locked/protected).
`fDupPath`	A file of that name already exists.
`fTMFOpen`	Too many files open.
`fIOErr`	Unspecified I/O error occurred.

FCreateAlways

syntax `MgErr` `FCreateAlways(fdp, path, permissions, openMode, denyMode, group);`

`FCreateAlways` creates a file with the name and location specified by **path** and with the specified **permissions**, and opens the file for writing and reading, as specified by **openMode**. If the file already exists, this function opens and truncates the file.

You can use **denyMode** to control concurrent access to the file from within LabVIEW. The **group** parameter allows you to assign the file to a UNIX group; under Windows or Macintosh, **group** is ignored.

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

Note: Before attempting to call this function, make sure that you understand how to use the fdp parameter. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about this parameter.

Parameter	**Type**	**Description**
fdp	`File *`	Address at which `FCreateAlways` stores the file descriptor for the new file. If `FCreateAlways` fails, it stores 0 in the address **fdp**.See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.
path	`Path`	Path of the file that you want to create.
permissions	`int32`	Permissions to assign to the new file For a description of permissions, see the topic file permissions.
openMode	`int32`	See FMOpen for a description of **openMode**.
denyMode	`int32`	See FMOpen for a description of **denyMode**.
group	`PStr`	UNIX group you want to assign to the new file.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fIsOpen`	File is already open for writing. This error is returned only on the Macintosh and the Sun. The PC **returns** `fIOErr` when the file is already open for writing.
`fNoPerm`	Access denied (something is locked/protected).
`fDupPath`	A file of that name exists.
`fTMFOpen`	Too many files open.
`fIOErr`	Unspecified I/O error occurred.

FMClose

syntax `MgErr` `FMClose(fd);`

`FMClose` closes the file associated with the file descriptor **fd**.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file you want to close.

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

Error	Description
`mgArgErr`	Not a valid file descriptor.
`fIOErr`	Unspecified I/O error occurred.

FMOpen

syntax `MgErr` `FMOpen(fdp, path, openMode, denyMode);`

Note: Before attempting to call this function, make sure that you understand how to use the fdp parameter. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about this parameter.

`FMOpen` opens a file with the name and location specified by **path** for writing and reading, as specified by **openMode**.

With the **denyMode** parameter, you control concurrent access to the file from within LabVIEW.

If this function opens the file, the resulting file descriptor is stored in the address referred to by **fdp**. If an error occurs, 0 is stored in the address referred to by **fdp** and the error is returned.

Parameter	Type	Description
fdp	`File *`	Address at which FMOpen stores the file descriptor for the opened file. If the function fails, FMOpen stores 0 in the address **fdp**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.
path	`Path`	Path of the file that you want to open.
openMode	`int32`	Access mode to use in opening the file. Can have the following values, which are defined in the file `extcode.h`. • `openReadOnly`: Open for reading.

		• `openWriteOnly`: Open for writing; file is not truncated (data is not removed). On the Macintosh, this mode provides true write-only access to files. On a PC or a UNIX system, LabVIEW I/O functions are built in the C standard I/O library, with which you have write-only access to a file only if you are truncating the file or making the access append-only. Therefore, this mode actually allows both read and write access to files on a PC or UNIX system. • `openReadWrite`: Open for both reading and writing. • `openWriteOnlyTruncate`: Open for writing; truncates the file.
denyMode	`int32`	Mode that determines what level of concurrent access to the file is allowed. Can have the following values, which are defined in the file extcode.h. • `denyReadWrite`: Prevents others from reading from and writing to the file while it is open. • `denyWriteOnly`: Prevents others from writing to the file only while it is open • `denyNeither`: allows others to read from and write to the file while it is open.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fIsOpen`	File is already open for writing. This error is returned only on the Macintosh and the Sun. The PC **returns** `fIOErr` when the file is already open for writing.
`fNotFound`	File not found.
`fTMFOpen`	Too many files open.
`fIOErr`	Unspecified I/O error occurred.

FMRead

syntax `MgErr` `FMRead(fd, inCount, outCountp, buffer);`

`FMRead` reads **inCount** bytes from the file specified by the file descriptor **fd**. The function starts from the current position mark (see the `FMSeek` and `FMTell` functions), and reads the data into memory, starting at the address specified by **buffer**.

The function stores the actual number of bytes read in ***outCountp**. The number of bytes can be less than **inCount** if the function encounters end-of-file before reading **inCount** bytes. The number of bytes will be zero if any other error occurs.

Parameter	Type	Description
fd	`File`	File descriptor associated with the file from which you want to read.
inCount	`int32`	Number of bytes you want to read.
outCountp	`int32 *`	Address at which `FMRead` stores the number of bytes read. `FMRead` will not store any value if `NULL` is passed.

		See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.
buffer	`UPtr`	Address where `FMRead` will store the data.

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

Error	**Description**
`mgArgErr`	Not a valid file descriptor or **inCount** < 0.
`fEOF`	EOF encountered.
`fIOErr`	Unspecified I/O error occurred.

FMWrite

syntax `MgErr` `FMWrite(fd, inCount, outCountp, buffer);`

`FMWrite` writes **inCount** bytes from memory, starting at the address specified by **buffer**, to the file specified by the file descriptor **fd**, starting from the current position mark (see the `FMSeek` and `FMTell` functions).

The function stores the actual number of bytes written in ***outCountp**. The number of bytes stored can be less than **inCount** if an `fDiskFull` error occurs before the function writes **inCount** bytes. The number of bytes stored will be zero if any other error occurs.

Parameter	**Type**	**Description**
fd	`File`	File descriptor associated with the file to which you want to write.
inCount	`int32`	Number of bytes you want to write.
outCountp	`int32 *`	Address at which `FMWrite` stores the number of bytes actually written. `FMWrite` will not store any value if `NULL` is passed. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview* in the *Code Interface Reference Manual* for more information about using this parameter.
buffer	`UPtr`	Address of the data you want to write.

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

Error	**Description**
`mgArgErr`	Not a valid file descriptor or **inCount** < 0.
`fDiskFull`	Out of space.
`fNoPerm`	Access denied.
`fIOErr`	Unspecified write error occurred.

File Range, Locking

FLockOrUnlockRange

Click here to view a list of all File Manager Functions.

FLockOrUnlockRange

syntax MgErr FLockOrUnlockRange(fd, mode, offset, count, lock);

FLockOrUnlockRange locks or unlocks a section of a file.

Parameter	Type	Description
fd	File	File descriptor associated with the file.
mode	int32	Position in the file relative to which FLockOrUnlockRange determines the first byte to lock or unlock. If **mode** is fStart, the first byte to lock or unlock is located **offset** bytes from the start of the file (**offset** must be greater than or equal to 0). If **mode** is fCurrent, the first byte to lock or unlock is located **offset** bytes from the current position mark (**offset** can be positive, 0, or negative). If **mode** is fEnd, the first byte to lock or unlock is located **offset** bytes from the end of the file (**offset** must be less that or equal to 0).
offset	int32	The position of the first byte to lock or unlock. The position is the number of bytes from the beginning of the file, the current position mark, or the end of the file, as determined by **mode**.
count	int32	Number of bytes to lock or unlock starting at the location specified by **mode** and **offset**.
lock	Bool32	A boolean that specifies whether FLockOrUnlockRange locks or unlocks a range of bytes. If lock is TRUE this functions locks a range; if FALSE the function unlocks a range.

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

Error	Description
fIOErr	Unspecified I/O error occurred.

File Refnums, Manipulating

Click here to view a list of all File Manager Functions.

FDisposeRefNum

syntax `MgErr FDisposeRefNum(refNum);`

FDisposeRefNum disposes of the specified file refnum.

Parameter	**Type**	**Description**
refNum	`LVRefNum`	File refnum of which you want to dispose.

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

Error	**Description**
`mgArgErr`	Invalid file refnum.

FIsARefNum

syntax `Bool32 FIsARefNum(refNum);`

`FIsARefNum` determines whether **refNum** is a valid file refnum.

Parameter	**Type**	**Description**
refNum	`LVRefNum`	File refnum whose validity you want to determine.

returns A boolean, which can have the following values for this function.

Value	**Description**
`TRUE`	File refnum has been created and not yet disposed.
`FALSE`	Otherwise.

FNewRefNum

syntax `MgErr FNewRefNum(path, fd, refNumPtr);`

`FNewRefNum` creates a new file refnum for an open file with the name and location specified by **path** and the file descriptor **fd**.

If the file refnum is created, the resulting file refnum is stored in the address referred to by **refNumPtr**. If an error occurs, `NULL` is stored in the address referred to by **refNumPtr** and the error is returned.

Parameter	**Type**	**Description**
path	`Path`	The path of the open file for which you wish to create a

		file refnum.
fd	`File`	The file descriptor of the open file for which you wish to create a file refnum.
refNumPtr	`LVRefNum *`	Address at which `FNewRefNum` stores the new file refnum. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview,* in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	**Description**
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.

FRefNumToFD

syntax `MgErr` `FRefNumToFD(refNum, fdp);`

`FRefNumToFD` gets the file descriptor associated with the specified file refnum.

If no error occurs, the resulting file descriptor is stored in the address referred to by **fdp**. If an error occurs, `NULL` is stored in the address referred to by **fdp** and the error is returned.

Parameter	**Type**	**Description**
refNum	`LVRefNum`	The file refnum whose associated file descriptor you wish to get.
fdp	`File *`	Address at which `FRefNumToFD` stores the file descriptor associated with the specified file refnum. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	**Description**
`mgArgErr`	Invalid file refnum.

FRefNumToPath

syntax `MgErr` `FRefNumToPath(refNum, path);`

`FRefNumToPath` gets the path associated with the specified file refnum, and stores the resulting path in the existing path, **path**.

If no error occurs, **path** is set to the path associated with the specified file refnum. If an error occurs, **path** is set to the canonical invalid path.

Parameter	**Type**	**Description**
refNum	`LVRefNum`	The file refnum whose associated path you wish to get.
path	`Path`	Path where `FRefNumToPath` stores the path associated with the specified file refnum. This path must already have been created.

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

Error	Description
mgArgErr	A bad argument was passed to the function. Verify path.
mFullErr	Insufficient memory.

Filenames and Patterns, Matching

FStrFitsPat

Click here to view a list of all File Manager Functions.

FStrFitsPat

syntax `Bool32 FStrFitsPat(pat, str, pLen, sLen);`

`FStrFitsPat` determines whether a filename, **str**, matches a pattern, **pat**.

Parameter	Type	Description
pat	`uChar *`	Pattern (string) to which filename is to be compared. The following characters have special meanings in the pattern. • `\` : The following character is literal, not treated as having a special meaning. A single backslash at the end of **pat** is the same as two backslashes. • `?` : Match any one character. • `*` : Match zero or more characters.
str	`uChar *`	Filename (string) to compare to pattern.
pLen	`int32`	Number of characters in **pat**.
sLen	`int32`	Number of characters in **str**.

returns `FStrFitsPat` **returns** `TRUE` if the filename fits the pattern; `FALSE` if otherwise.

Files and Directories, Moving and Deleting

FMove
FRemove

Click here to view a list of all File Manager Functions.

FMove

syntax `MgErr` `FMove(oldPath, newPath);`

`FMove` moves a file or renames it if the new path indicates the file is to remain in the same directory.

Parameter	Type	Description
oldPath	`Path`	Path of the file or directory you want to move.
newPath	`Path`	Path, including the name of the file or directory, where you want the file or directory to be moved.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fNotFound`	The original file could not be found.
`fNoPerm`	Access denied (file/directory/disk is locked/protected).
`fDiskFull`	Disk is full.
`fDupPath`	The new file already exists.
`fIsOpen`	The original file is open for writing.
`fTMFOpen`	Too many files open.
`mFullErr`	Insufficient memory.
`fIOErr`	Read, write, or unspecified I/O error occurred.

FRemove

syntax `MgErr` `FRemove(path);`

`FRemove` deletes a file or a directory. If an error occurs, this function does not remove the file or directory.

Parameter	Type	Description
path	`Path`	Path of the file or directory you want to delete.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fNotFound`	The file could not be found.
`fNoPerm`	Access denied (file/directory/disk is locked/protected).
`fIsOpen`	File is open or directory is not empty.
`fIOErr`	Unspecified I/O error occurred.

Files, Copying

FCopy

Click here to view a list of all File Manager Functions.

FCopy

syntax `MgErr` `FCopy(oldPath, newPath);`

`FCopy` copies a file, preserving the type, creator, and access rights. The file to be copied must not be open. If an error occurs, the new file is not created.

Parameter	Type	Description
oldPath	`Path`	Path of the file you want to copy.
newPath	`Path`	Path, including filename, where you want the new file to be stored.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`fNotFound`	The original file could not be found.
`fNoPerm`	Access denied (file/directory/disk is locked/protected).
`fDiskFull`	Disk is full.
`fDupPath`	The new file already exists.
`fIsOpen`	The original file is open for writing.
`fTMFOpen`	Too many files open.
`mFullErr`	Insufficient memory.
`fIOErr`	Read, write, or unspecified I/O error occurred

Paths, Comparing

FIsAPath
FIsAPathOrNotAPath
FIsEmptyPath
FPathCmp

Click here to view a list of all File Manager Functions.

FIsAPath

syntax `Bool32 FIsAPath(path);`

`FIsAPath` determines whether **path** is a valid path.

Parameter	Type	Description
path	`Path`	Path whose validity you want to determine.

returns A boolean, which can have the following values for this function.

Value	Description
`TRUE`	Path is well formed and type is absolute or relative.
`FALSE`	Otherwise.

FIsAPathOrNotAPath

syntax `Bool32 FIsAPathOrNotAPath(path);`

`FIsAPathOrNotAPath` determines whether **path** is a valid path or the canonical invalid path.

Parameter	Type	Description
path	`Path`	Path whose validity you want to determine.

returns A boolean, which can have the following values for this function.

Value	Description
`TRUE`	Path is well formed, and type is absolute, relative, or not a path.
`FALSE`	Otherwise.

FIsEmptyPath

syntax `Bool32 FIsEmptyPath(path);`

`FIsEmptyPath` determines whether **path** is a valid empty path.

Parameter	Type	Description
path	`Path`	Path whose validity and emptiness you want to determine.

returns A boolean, which can have the following values for this function.

Value	Description
`TRUE`	Path is well formed and empty, and type is absolute or relative.
`FALSE`	Otherwise.

FPathCmp

syntax `int32 FPathCmp(lsp1, lsp2);`

`FPathCmp` compares the two specified paths.

Parameter	Type	Description
lsp1	`Path`	First path to compare.
lsp2	`Path`	Second path to compare.

returns `int32`, which can have the following values for this function.

Value	Description
`-1`	Paths are of different types (for example, one is absolute and the other is relative).
`0`	Paths are identical.
`n+1`	Paths have the same first *n* components, but are not identical.

Paths, Converting to and from Other Representations

FArrToPath
FFlattenPath
FPathToArr
FPathToAZString
FPathToDSSString
FStringToPath
FTextToPath
FUnFlattenPath

Click here to view a list of all File Manager Functions.

FArrToPath

syntax `MgErr FArrToPath(arr, relative, path);`

`FArrToPath` converts a specified one-dimensional LabVIEW array of strings to a path of the type specified by **relative**. Each string in the specified array is converted in order into a component name of the resulting path.

If no error occurs, **path** is set to a path whose component names are the strings in **arr**. If an error occurs, **path** is set to the canonical invalid path.

Parameter	Type	Description
arr	`UHandle`	The (DS) handle containing the array of strings which you wish to convert to a path.
relative	Bool32	If **relative** is `TRUE`, then the resulting path is relative; otherwise, the resulting path is absolute.
path	`Path`	Path where `FArrToPath` stores the resulting path. This path must already have been allocated.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.

FFlattenPath

syntax `int32 FFlattenPath(p, fp);`

`FFlattenPath` converts a path into a flat form that you can use to write the path as information to a file. The function stores the resulting flat path in a pre-allocated buffer and **returns** the number of bytes.

You can determine the size needed for the flattened path by passing `NULL` for **fp**, in which case the function **returns** the necessary size without writing anything into the location pointed to by fp.

Parameter	Type	Description
p	`Path`	Path you want to flatten.
fp	`UPtr`	Address in which `FFlattenPath` stores the resulting

		flattened path. If this value is NULL, FFlattenPath does not write anything to this address, but does return the size that the flattened path would require. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

returns int32, indicating the number of bytes required to store the flattened path.

FPathToArr

syntax `MgErr FPathToArr(path, relativePtr, arr);`

FPathToArr converts a specified path to a one-dimensional LabVIEW array of strings and determines whether the specified path is relative. Each component name of the specified path is converted in order into a string in the resulting array.

If no error occurs, **arr** is set to an array of strings containing the component names of **path**. If an error occurs, **arr** is set to an empty array.

Parameter	Type	Description
path	Path	The path which you wish to convert to an array of strings.
relativePtr	Bool32 *	Address at which to store a boolean value telling whether the specified path is relative. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.
arr	UHandle	(DS) Handle where FPathToArr stores the resulting array of strings. This handle must already have been allocated.

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

Error	Description
mgArgErr	Badly formed path or unallocated array.
mFullErr	Insufficient memory.

FPathToAZString

syntax `MgErr FPathToAZString(p, txt);`

FPathToAZString converts a specified path to an LStr and stores the string as an application zone handle. The LStr contains the platform-specific syntax for the path.

Parameter	Type	Description
p	Path	Path that you want to convert to a string.
txt	LStrHandle *	Address at which FPathToAZString stores the resulting string. If the value at **txt** is nonzero, the function assumes that it is a valid handle, resizes the handle, fills in its value, and stores the handle at the address referred to by txt. See the *Pointers as Parameters* section of Chapter 1,

CIN Overview, in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error occurred.

FPathToDSString

syntax `MgErr FPathToDSString(p, txt);`

`FPathToDSString` converts a specified path to an `LStr` and stores the string as a data space zone handle. The `LStr` contains the platform-specific syntax for the path.

Parameter	Type	Description
p	`Path`	Path that you want to convert to a string.
txt	`LStrHandle *`	Address at which FPathToDSString stores the resulting string. If the value at **txt** is nonzero, the function assumes that it is a valid handle, resizes the handle, fills in its value, and stores the handle at the address referred to by txt. See the *Pointer as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.
`fIOErr`	Unspecified I/O error occurred.

FStringToPath

syntax `MgErr FStringToPath(text, p);`

`FStringToPath` creates a path from an `LStr`. The `LStr` contains the platform-specific syntax for a path.

Parameter	Type	Description
text	`LStrHandle`	String that contains the path in platform-specific syntax.
p	`Path *`	Address at which `FStringToPath` stores the resulting path. If the value at **p** is non-zero, the function assumes that it is a valid path, resizes the path, and fills in its value. If the value at **p** is zero (NULL), the function creates a new path, fills in its value, and stores the path at the address referred to by **p**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	Description
mFullErr	Insufficient memory.

FTextToPath

syntax `MgErr FTextToPath(text, tlen, *p);`

FTextToPath creates a path from a string (at the address **text**) that represents a path in the platform-specific syntax for a path.

Parameter	Type	Description
text	UPtr	String that contains the path in platform-specific syntax.
tlen	int32	Number of characters in **text**.
p	Path *	Address at which FTextToPath stores the resulting path. If the value at **p** is non-zero, the function assumes that it is a valid path, resizes the path, and fills in its value. If the value at **p** is zero (NULL), the function creates a new path, fills in its value, and stores the path at the address referred to by **p**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	Description
mFullErr	Insufficient memory.

FUnFlattenPath

syntax `int32 FUnFlattenPath(fp, pPtr);`

FUnFlattenPath converts a flattened path (created using FFlattenPath) into a path.

Parameter	Type	Description
fp	UPtr	Pointer to the flattened path you want to convert to a path.
pPtr	Path *	Address at which FUnFlattenPath stores the resulting path. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

returns The number of bytes the function interpreted as a path.

Paths, Creating

Click here to view a list of all File Manager Functions.

FAddPath

syntax `MgErr` `FAddPath(basePath, relPath, newPath);FAddPath` creates an absolute path by appending a relative path to an absolute path

Note: You can pass in the same path variable for the new path that you use for the `basePath` **or** `relPath`**. Thus, the following three variations for calling this function work.**

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

Parameter	Type	Description
basePath	`Path`	Absolute path to which you want to append a relative path.
relPath	`Path`	Relative path you want to append to the existing base path.
newPath	`Path`	Path returned by `FAddPath`.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.

FAppendName

syntax `MgErr` `FAppendName(path, name);`

`FAppendName` appends a file or directory name to an existing path.

Parameter	Type	Description
path	`Path`	Base path to which you want to append a new file or directory name. `FAppendName` **returns** the resulting

		path in this parameter.
name	`PStr`	File or directory name that you want to append to the existing path.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.

FAppPath

syntax `MgErr FAppPath(p);`

`FAppPath` determines the path to the currently executing LabVIEW application.

Parameter	Type	Description
p	`Path`	Path in which `FAppPath` stores the path to the currently executing LabVIEW application. **p** must already be an allocated path.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.
`fNotFound`	File not found.
`fIOErr`	Unspecified I/O error occurred.

FEmptyPath

syntax `Path` `FEmptyPath(p);`

`FEmptyPath` makes an empty absolute path. Making a path an empty absolute path is not the same as disposing the path.

Parameter	Type	Description
p	`Path`	Path allocated by `FEmptyPath`. If **p** is `NULL`, `FEmptyPath` allocates a new path and **returns** the value. If **p** is a path, the existing path is set to be an empty path, and the new **p** is returned.

returns The resulting path; if **p** was not `NULL`, the return value is the same emptyabsolute path as **p**. If an error occurs, `NULL` is returned.

FMakePath

syntax `Path` `FMakePath(path, type, [volume, directory, directory, ..., name,] NULL);`

The brackets indicate that the volume, directory, and name parameters are optional.

`FMakePath` creates a new path. If path is `NULL`, the function allocates and **returns** a new path. Otherwise, path is set to the new path, and path is returned. If an error occurs, or the path is not specified correctly, `NULL` is returned.

When you are finished using a path, you should dispose of it using `FDisposePath`.

Parameter	Type	Description
path	`Path`	Parameter in which `FMakePath` **returns** the newly created path if **path** is not `NULL`.
type	`int32`	Type of path to create. If type is `fAbsPath`, the new path will be absolute. If type is `fRelPath`, the new path will be relative.
vol	`PStr`	Pascal string containing a legal volume name. An empty string means go up a level in the path hierarchy. This parameter is optional, and is only used for absolute paths on Macintosh or Windows platforms.
directory	`PStr`	Pascal string containing a legal directory name. An empty string means go up a level in the path hierarchy. Parameter is optional.
name	`PStr`	File or directory name. An empty string means go up a level in the path hierarchy. Parameter is optional.
NULL	`PStr`	Marker indicating the end of the path.

returns The resulting **path**; if you specified **path**, the return value is the same path as **path**. If an error occurs, `NULL` is returned.

FNotAPath

syntax `Path FNotAPath(p);`

`FNotAPath` creates a path that is the canonical invalid path.

Parameter	Type	Description
p	`Path`	Path allocated by `FNotAPath`. If **p** is `NULL`, `FNotAPath` allocates a new canonical invalid path and **returns** the value. If **p** is a path, the existing path is set to the canonical invalid path, and the new **p** is returned.

returns The resulting path. If **p** was not `NULL`, the return value is the same canonical invalid path as **p**. If an error occurs, `NULL` is returned.

FRelPath

syntax `MgErr FRelPath(startPath, endPath, relPath);`

`FRelPath` computes a relative path between two absolute paths.

Note: You can pass in the same path variable for the new path that you use for the `startPath` **or** `relPath`**. Thus, the following three variations for calling this function work.**

```
FRelPath(startPath, endPath, relPath);
/* the relative path is returned in a third path variable */
FRelPath(startPath, endPath, startPath);
```

```
/* the new path writes over the old startPath */
FRelPath(startPath, endPath, endPath);
/* the new path writes over the old endPath */
```

Parameter	Type	Description
startPath	`Path`	Absolute path from which you want the relative path to be computed.
endPath	`Path`	Absolute path to which you want the relative path to be computed.
relPath	`Path`	Path returned by `fAddPath`.

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

Error	Description
`mgArgErr`	A bad argument was passed to the function. Verify path.
`mFullErr`	Insufficient memory.

Paths, Disposing

FDisposePath

Click here to view a list of all File Manager Functions.

FDisposePath

syntax `MgErr FDisposePath(p);`

`FDisposePath` disposes of the specified path.

Parameter	**Type**	**Description**
p	`Path`	Path you want to dispose of.

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

Error	**Description**
`mZoneErr`	Invalid path**.**

Paths, Duplicating

FPathCpy
FPathToPath

Click here to view a list of all File Manager Functions.

FPathCpy

syntax `MgErr` `FPathCpy(dst, src);`

`FPathCpy` duplicates the path specified by **src**, and stores the resulting path in the existing path, **dst**.

Parameter	Type	Description
dst	`Path`	Path where `FPathCpy` places the resulting duplicate path. This path must already have been created.
src	`Path`	Path that you want to duplicate.

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

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

FPathToPath

syntax `MgErr` `FPathToPath(p);`

`FPathToPath` duplicates the specified path and **returns** the new path in the same variable.

Parameter	Type	Description
p	`Path *`	Address of path to duplicate. Variable to which `FPathToPath` **returns** the resulting path. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

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

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

Path, Extracting Information

FDepth
FDirName
FNameFNamePtr
FVolName

Click here to view a list of all File Manager Functions.

FDepth

syntax `int32 FDepth(path);`

`FDepth` computes the depth (number of component names) of a specified path.

Parameter	Type	Description
path	Path	Path whose depth you want to determine.

returns `int32` indicating the depth of the specified path, which can have the following values for this function.

Value	Description
`-1`	Badly formed path.
`0`	Path is the root directory.
`1`	Path is in the root directory.
`2`	Path is in a subdirectory of the root directory, one level from the root directory.
`n-1`	Path is *n-2* levels from the root directory.
`n`	Path is *n-1* levels from the root directory.

FDirName

syntax `MgErr FDirName(path, dir);`

`FDirName` creates a path for the parent directory of a specified path.

Note: You can pass in the same path variable for the parent path that you use for path. Thus, the following variations for calling this function work.

```
err = FDirName(path, dir);
/* the parent path is returned in a second path variable */
err = FDirName(path, path);
/* the parent path writes over the existing pathÊ*/
```

Parameter	Type	Description
path	`Path`	Path whose parent path you want to determine.
dir	`Path`	Parameter in which `FDirName` stores the parent path.

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

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

FName

syntax `MgErr` `FName(path, name);`

`FName` copies the last component name of a specified path into a string handle and resizes the handle as necessary.

Parameter	Type	Description
path	`Path`	Path whose last component name you want to determine.
name	`StringHandle`	Handle in which FName **returns** the last component name as a Pascal string.

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

Error	Description
`mgArgErr`	Badly formed path or path is root directory.
`mFullErr`	Insufficient memory.

FNamePtr

syntax `MgErr` `FNamePtr(path, name);`

`FNamePtr` copies the last component name of a specified path to the address specified by name. This routine does not allocate space for the returned data, so name must specify allocated memory of sufficient size to hold the component name.

Parameter	Type	Description
path	`Path`	Path whose last component name you want to determine.
name	`PStr`	Address at which `FNamePtr` stores the last component name as a Pascal string. This address must specify allocated memory of sufficient size to hold the name. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

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

Error	Description
`mgArgErr`	Badly formed path or path is root directory.
`mFullErr`	Insufficient memory.

FVolName

syntax `MgErr` `FVolName(path, vol);`

`FVolName` creates a path for the volume of a specified absolute path by removing all but the first component name from **path**.

Note: **You can pass in the same path variable for the volume path that you use for `path`. Thus, the following variations for calling this function work.**

```
err = FVolName(path, vol);
/* the parent path is returned in a second path variable */
err = FVolName(path, path);
/* the parent path writes over the existing pathÊ*/
```

Parameter	Type	Description
path	`Path`	Path whose volume path you want to determine.
vol	`Path`	Parameter in which `FVolName` stores the volume path.

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

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

Path Type, Determining

FGetPathType
FIsAPathOfType
FSetPathType

Click here to view a list of all File Manager Functions.

FGetPathType

syntax `MgErr FGetPathType(path, typePtr)`

`FGetPathType` **returns** the type (relative, absolute, or not a path) of the specified path.

Parameter	**Type**	**Description**
path	`Path`	Path whose type you want to determine.
typePtr	`int32 *`	Address at which `FGetPathType` stores the type. ***typePtr** can have the following values: • `fAbsPath`: The path is an absolute path. • `fRelPath`: The path is a relative path. • `fNotAPath`: The path is the canonical invalid path or an error occurred. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

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

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

FIsAPathOfType

syntax `Bool32 FIsAPathOfType(path, ofType);`

`FIsAPathOfType` determines whether the specified path is a valid path of the specified type (relative or absolute).

Parameter	**Type**	**Description**
path	`Path`	Path that you want to compare to the specified type.
ofType	`int32`	Type that you want to compare to the path's type. **type** can have the following values: • `fAbsPath`: Compare the path's type to absolute. • `fRelPath`: Compare the path's type to relative.

returns A boolean, which can have the following values for this function.

Values	**Description**
`TRUE`	Path is well formed and type is identical to **ofType**.
`FALSE`	Otherwise.

FSetPathType

syntax `MgErr` `FSetPathType(path, type);`

`FSetPathType` changes the type of the specified path (which must be a valid path) to the specified type (relative or absolute).

Parameter	**Type**	**Description**
path	`Path`	Path whose type you want to change.
type	`int32`	New type that you want the path to have. **type** can have the following values: • `fAbsPath`: The path is an absolute path. • `fRelPath`: The path is a relative path.

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

Error	**Description**
`mgArgErr`	Badly formed path or invalid type.

File/Directory Information Record

Several routines in the file manager work with a data structure that defines the attributes of a file or directory. The following list gives the file/directory information record.

```
typedef struct {
   int32   type;        * system specific file type-- 0 for
                        directories */
   int32   creator;     * system specific file creator-- 0
                        for folders (on Mac only)*/
   int32   permissions; * system specific file access
                        rights */
   int32   size;        /* file size in bytes (data fork on
                        Mac) or entries in directory*/
   int32   rfSize;      /* resource fork size (on Mac only)
                        */
   uInt32  cdate;       /* creation date: seconds since
                        system reference time */
   uInt32  mdate;       /* last modification date: seconds
                        since system ref time */
   Bool32  folder;      /* indicates whether path refers to
                        a folder */
   Bool32  isInvisible; /* indicates whether file is
                        visible in File Dialog (on Mac
                        only)*/
   Point   location;    /* system specific desktop
                        geographical location (on Mac
                        only)*/
   Str255  owner;       /* owner (in pascal string form) of
                        file or folder */
   Str255  group;       /* group (in pascal string form) of
                        file or folder */
   }       FInfoRec, *FInfoPtr;
```

File Type Record

The file type record is:

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

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

```
#define kIsFile 0x01
#define kRecognizedType 0x02
#define kIsLink 0x04
#define kFIsInvisible 0x08
```

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

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

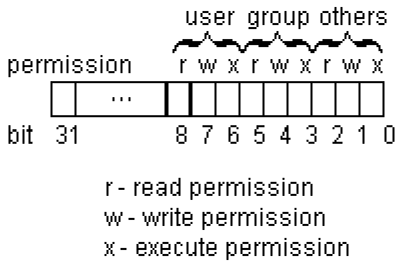
Path Data Type

The file manager defines the `Path` data type for use in describing paths to files and directories. The data structure for the `Path` data type is private. You use file manager routines to create and manipulate `Paths`.

File Permissions

The file manager uses the `int32` data type to describe permissions for files and directories. The manager uses only the least significant nine bits of the `int32`.

On a UNIX computer, the nine bits of permissions correspond exactly to nine UNIX permission bits governing read, write, and execute permissions for user, group, and others. Permission bits on a UNIX system are represented in the following illustration.



On the PC, permissions are ignored for directories. For files, only bit 7 (the UNIX user write permission bit) is used. If this bit is clear, the file is read-only. Otherwise, you can write to the file.

On the Macintosh, all nine bits are used for directories (folders). The bits which control read, write, and execute permissions, respectively, on a UNIX system are used to control See Files, Make Changes, and See Folders access rights, respectively, on the Macintosh. For files, only bit 7 (the UNIX user write permission bit) is used. If this bit is clear, the file is locked. Otherwise, the file is not locked.

Volume Information Record

The volume information record is:

```
typedef struct {
   int32   size;    /* size in bytes of a kuhvkjhgvku volume
                      */
   int32   used;    /* number of bytes used on volume */
   int32   free;    /* number of bytes available for use on
                      volume */
}          VInfoRec;
```

Memory Manager Functions

This topic contains descriptions of the memory manager functions that perform the following operations:

Handle and Pointer Verification
Handles, Allocating and Releasing
Handles, Manipulating Properties
Memory Utilities
Memory Zone Utilities
Pointers, Allocating and Releasing

Handle and Pointer Verification

Click here to view a list of all Memory Manager Functions.

AZCheckHandle/DSCheckHandle

syntax

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

`XXCheckHandle` verifies that the specified handle is really a handle. If the handle is not a real handle, this function returns mZoneErr.

Parameter	Type	Description
h	`UHandle`	Handle to verify.

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

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

AZCheckPtr/DSCheckPtr

syntax

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

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

Parameter	Type	Description
p	`UPtr`	Pointer to verify.

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

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

Handles, Allocating and Releasing

Click here to view a list of all Memory Manager Functions.

AZDisposeHandle/DSDisposeHandle

syntax

```
MgErrAZ          DisposeHandle(h);
MgErrDS          DisposeHandle(h);
```

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

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

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

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

AZEmptyHandle/DSEmptyHandle

syntax

```
MgErrAZ          EmptyHandle(h);
MgErrDS          EmptyHandle(h);
```

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

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

Parameter	Type	Description
h	UHandle	Handle to empty.

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

Error	Description
noErr	No error.
mZone	ErrHandle or pointer not in specified zone.

AZGetHandleSize/DSGetHandleSize

syntax

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

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

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

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

AZNewHandle/DSNewHandle

syntax

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

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

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

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

AZNewHClr/DSNewHClr

syntax

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

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

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

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

AZReallocHandle/DSReallocHandle

syntax

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

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

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

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

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

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

AZRecoverHandle/DSRecoverHandle

syntax

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

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

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

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

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

AZSetHandleSize/DSSetHandleSize

syntax

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

`XXSetHandleSize` changes the size of the block of memory referenced by the specified handle.

While LabVIEW arrays are stored in DS handles, you should not use this function to resize array handles. Many platforms have memory alignment requirements that make it difficult to determine the correct size for the resulting array. Instead, you should use either `NumericArrayResize` or `SetCINArraySize`, which are described in the *Resizing Arrays and Strings* section of Chapter 2, *CIN Parameter Passing* in the *Code Interface Reference Manual*. You should not use these functions on a locked handle.

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

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

Error	Description
`noErr`	No error.

`mFullErr`	Not enough memory to perform operation.
`mZoneErr`	Handle or pointer not in specified zone.

AZSetHSzClr/DSSetHSzClr

syntax

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

`XXSetHSzClr` changes the size of the block of memory referenced by the specified handle and sets any new memory to zero. You should not use this function on a locked handle.

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

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

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

Handles, Manipulating Properties

Click here to view a list of all Memory Manager Functions.

AZHLock

syntax `MgErr` `AZHLock(h);`

`AZHLock` locks the memory referenced by the application zone handle **h** so that the memory cannot move. This means the memory manager cannot move the block of memory to which the handle refers.

Do not lock handles more than necessary; it interferes with efficient memory management. Also, do not enlarge a locked handle.

Parameter	Type	Description
h	`UHandle`	Application zone handle to lock.

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

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

AZHPurge

syntax `void` `AZHPurge(h);`

`AZHPurge` marks the memory referenced by the application zone handle **h** as purgeable. This means that in tight memory conditions the memory manager can perform an `AZEmptyHandle` on **h.** Use `AZReallocHandle()` to reuse a handle if the manager purges it.

If you mark a handle as purgeable, check the handle before using it to see if it has become an empty handle.

Parameter	Type	Description
h	`UHandle`	Application zone handle to mark as purgeable.

AZHNoPurge

syntax `void` `AZHNoPurge(h);`

`AZHNoPurge` marks the memory referenced by the application zone handle **h** as unpurgeable.

Parameter	Type	Description
h	`UHandle`	Application zone handle to mark as unpurgeable.

AZHUnlock

syntax `MgErr AZHUnlock(h);`

AZHUnlock unlocks the memory referenced by the application zone handle **h** so that it can be moved. This means that the memory manager can move the block of memory to which the handle refers if other memory operations need space.

Parameter	Type	Description
h	UHandle	Application zone handle to unlock.

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

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

Memory Utilities

Click here to view a list of all Memory Manager Functions.

AZHandAndHand/DSHandAndHand

syntax

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

`XXHandAndHand` appends the data referenced by **h1** to the end of the memory block referenced by **h2**.

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

Parameter	Type	Description
h1	`UHandle`	Source of data to append to **h2**.
h2	`UHandle`	Initial handle, to which the data of **h1** is appended.

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

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

AZHandToHand/DSHandToHand

syntax

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

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

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

Parameter	Type	Description
hp	`UHandle`	Pointer to handle to duplicate. A pointer to the resulting handle is returned in this parameter. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the

Code Interface Reference Manual for more information about using this parameter.

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

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

AZPtrAndHand/DSPtrAndHand

syntax

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

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

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

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

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

AZPtrToHand/DSPtrToHand

syntax

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

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

Parameter	Type	Description
p	`UPtr`	Source of data to copy to the handle pointed to by **hp**.
hp	`UHandle`	Pointer to handle to duplicate. A pointer to the resulting handle is returned in this parameter. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter
size	`int32`	Number of bytes to copy from **p** to the new handle.

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

Error	Description
`noErr`	No error.
`mFullErr`	Not enough memory to perform operation.

AZPtrToXHand/DSPtrToXHand

syntax

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

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

Parameter	Type	Description
p	`UPtr`	Source of data to copy to the handle **h**.
h	`UHandle`	Destination handle.
size	`int32`	Number of bytes to copy from **p** to the existing handle.

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

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

ClearMem

syntax

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

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

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

MoveBlock

syntax

```
void            MoveBlock(ps, pd, size);
```

`MoveBlock` moves **size** bytes from one address to another. The source and destination memory blocks can overlap.

Parameter	Type	Description
ps	`UPtr`	Pointer to source.
pd	`UPtr`	Pointer to destination.
size	`int32`	Number of bytes to move.

SwapBlock

syntax

```
void            SwapBlock(ps, pd, size);
```

`SwapBlock` swaps **size** bytes between the section of memory referred to by **ps** and **pd**. The source and destination memory blocks should not overlap.

Parameter	Type	Description
ps	`UPtr`	Pointer to source.
pd	`UPtr`	Pointer to destination.

size	`int32`	Number of bytes to move.

Memory Zone Utilities

Click here to view a list of all Memory Manager Functions.

AZHeapCheck/DSHeapCheck

syntax

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

XXHeapCheck verifies that the specified heap is not corrupt. This function returns a zero for an intact heap and a nonzero value for a corrupt heap.

Parameter	Type	Description
d	Bool32	Dump extensive heap examination to auxiliary screen.

returns int32, which can contain the errors in the following list.

Value	Description
noErr	The heap is intact.
mCorruptErr	The heap is corrupt.

AZMaxMem/DSMaxMem

syntax

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

XXMaxMem returns the size of the largest block of contiguous memory available for allocation.

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

AZMemStats/DSMemStats

syntax

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

XXMemStats returns various statistics about the memory in a zone.

Parameter	Type	Description
msrp	MemStatRec	Returns statistics about the zone's free memory in a MemStatRec structure. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

A MemStatRec structure is defined as follows.

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

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

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

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

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

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

Pointers, Allocating and Releasing

Click here to view a list of all Memory Manager Functions.

AZDisposePtr/DSDisposePtr

syntax

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

`XXDisposePtr` releases the memory referenced by the specified pointer.

Parameter	Type	Description
p	`UPtr`	Pointer to dispose.

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

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

AZNewPClr/DSNewPClr

syntax

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

`XXNewPClr` creates a new pointer to a nonrelocatable block of memory of the specified size and initializes the memory to zero.

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

returns A pointer to a block of **size** bytes filled with zeros. Returns `NULL` if the allocation could not be performed.

AZNewPtr/DSNewPtr

syntax

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

`XXNewPtr` creates a new pointer to a nonrelocatable block of memory of the specified size.

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

returns A pointer to a block of **size** bytes. Returns `NULL` if the allocation could not be performed.

Support Manager Functions

This topic contains descriptions of the support manager functions that perform the following operations:

Byte Manipulation Operations
Mathematical Operations
String Manipulation
Utility Functions
Time Functions

Byte Manipulation Operations

Click here to view a list of all Support Manager Functions.

Cat4Chrs *Macro*

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

`Cat4Chrs` constructs an `int32` from four `uInt8s`, with the first parameter as the high byte and the last parameter as the low byte.

Parameter	Type	Description
a	`uInt8`	High order byte of the high word of the resulting `int32`.
b	`uInt8`	Low order byte of the high word of the resulting `int32`.
c	`uInt8`	High order byte of the low word of the resulting `int32`.
d	`uInt8`	Low order byte of the low word of the resulting `int32`.

returns The resulting `int32`.

GetALong *Macro*

syntax `int32` `GetALong(p);`

GetALong retrieves an `int32` from a `void` pointer. On the SPARCstation, this function can retrieve an `int32` at any address, even if the `int32` is not long word aligned.

Parameter	**Type**	**Description**
p	`void *`	Address from which you wish to read an `int32`.

returns `int32` stored at the specified address.

Hi16 *Macro*

syntax `int16` `Hi16(x);`

`Hi16` **returns** the high order `int16` of an `int32`.

Parameter	Type	Description
x	`int32`	`int32` of which you want to determine the high `int16`.

HiByte *Macro*

syntax `int8` `HiByte(x);`

`HiByte` **returns** the high order `int8` of an `int16`**.**

Parameter	Type	Description
x	`int16`	`int16` of which you want to determine the high `int8`.

HiNibble *Macro*

syntax `uInt8` `HiNibble(x);`

`HiNibble` **returns** the value stored in the high four bits of an `uInt8`**.**

Parameter	Type	Description
x	`uInt8`	`uInt8` whose high four bits you want to extract.

Lo16 *Macro*

syntax `int16` `Lo16(x);`

`Lo16` **returns** the low order `int16` of an `int32`.

Parameter	Type	Description
x	`int32`	`int32` of which you want to determine the low `int16`.

LoByte *Macro*

syntax `int8` `LoByte(x);`

LoByte **returns** the low order `int8` of an `int16`**.**

Parameter	Type	Description
x	`int16`	`int16` of which you want to determine the low `int8`.

Long *Macro*

syntax `int32` `Long(hi, lo);`

`Long` creates an `int32` from two `int16`**s.**

Parameter	Type	
hi	`int16`	High `int16` for the resulting `int32`.
lo	`int16`	Low `int16` for the resulting `int32`.

returns The resulting `int32`.

LoNibble *Macro*

syntax `uInt8 LoNibble(x);`

`LoNibble` **returns** the value stored in the low four bits of an `uInt8`**.**

Parameter	Type	Description
x	`uInt8`	`uInt8` whose low four bits you want to extract.

Offset *Macro*

syntax `int16 Offset(type, field);`

`Offset` **returns** the offset of the specified field within the structure called **type**.

Parameter	Type	Description
type	-	Structure that contains field.
field	-	Field whose offset you want to determine.

returns An offset as an `int16`.

SetALong *Macro*

syntax `void SetALong(p,x);`

`SetALong` stores an `int32` at the address specified by a void pointer. On the SPARCstation, this function can retrieve an `int32` at any address, even if it is not long word aligned.

Parameter	Type	Description
p	`void *`	Address at which you want to store an `int32`. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.
x	`int32`	Value that you want to store at the specified address.

Word*Macro*

syntax `int16 Word(hi, lo);`

`Word` creates an `int16` from two `int8`s**.**

Parameter	Type	Description
hi	`int8`	High `int8` for the resulting `int16`.
lo	`int8`	Low `int8` for the resulting `int16`.

returns The resulting `int16`.

Mathematical Operations

Abs
Max
Min
Pin
RandomGen

Click here to view a list of all Support Manager Functions.

In addition to the mathematical operations documented in this topic, LabVIEW supports a number of other mathematical functions. These functions are implemented as defined in *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie. Table 8.1 lists the prototypes for these functions.

Mathematical Functions Supported by LabVIEW

```
double atan(double);
double cos(double);
double exp(double);
double fabs(double);
double log(double);
double sin(double);
double sqrt(double);
double tan(double);
double acos(double);
double asin(double);
double atan2(double, double);
double ceil(double);
double cosh(double);
double floor(double);
double fmod(double, double);
double frexp(double, int *);
double ldexp(double, int);
double log10(double);
double modf(double, double *);
double pow(double, double);
double sinh(double);
double tanh(double);
```

For THINK C Users

To link the math functions when using THINK C, you need to add additional files to your project. You can link a modified version of an ANSI library provided by THINK C. The ANSI library must be modified to reference its globals from A4 instead of A5; this process is explained in the THINK C documentation in the section concerning building code resources (the section has different names in the various THINK C versions).

To make such a library, make a copy of the ANSI-A4 project (shipped with THINK C), and name it ANSI-A4 copy (or any unique name). Add the math.c file (shipped with THINK C) to ANSI-A4 copy, and then select **Build Library**... under the **Project** menu. Name your new library mathlib (or any unique name). Adding mathlib to your CIN project makes it possible for your math functions to link.

Abs

syntax `int32 Abs(n);`

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

Parameter	Type	Description
n	int32	int32 whose absolute value you want to find.

Max

syntax `int32 Max(n,m);`

Max **returns** the maximum of the two specified int32s.

Parameter	Type	Description
n,m	int32	int32s whose maximum value you want to determine.

Min

syntax `int32 Min(n,m);`

Min returns the minimum of the two specified int32s.

Parameter	Type	Description
n,m	int32	int32s whose minimum value you want to determine.

Pin

syntax `int32 Pin(i,low,high);`

Pin returns **i** coerced to fall within the range from **low** to **high** inclusive.

Parameter	Type	Description
i	int32	Value you want to coerce to the specified range.
n	int32	Low value of the range to which you want to coerce **i**.
m	int32	High value of the range to which you want to coerce **i**.

returns i coerced to the specified range.

RandomGen

syntax `void RandomGen(xp);`

RandomGen generates a random number between 0 and 1 and stores it at **xp**.

Parameter	Type	Description
xp	`float64 *`	Location to store the resulting double-precision floating-point random number. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

String Manipulation

BlockCmp
CPStrBuf
CPStrCmp
CPStrIndex
CPStrInsert
CPStrLen
CPStrRemove
CPStrReplace
CPStrSize
CToPStr
FileNameCmp
FileNameIndCmp
FileNameCmp
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

Click here to view a list of all Support Manager Functions.

BlockCmp

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

`BlockCmp` compares two blocks of memory to determine whether one is less than, the same as, or greater than the other.

Parameter	**Type**	**Description**
p1	`UPtr`	Pointer to a block of memory.
p2	`UPtr`	Pointer to a block of memory.
numBytes	`int32`	Number of bytes to compare.

returns A negative number, zero, or a positive number if **s1** is less than, the same as, or greater than **s2**.

CPStrBuf *Macro*

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

`CPStrBuf` **returns** the address of the first string in a concatenated list of Pascal strings (that is, the address of `sp->str`).

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

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

CPStrCmp

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

`CPStrCmp` lexically compares two concatenated lists of Pascal strings to determine whether one is less than, the same as, or greater than the other. This comparison is case sensitive, and the function compares the lists as if they were one string.

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

returns `<0, 0,` or `>0` if **s1** is less than, the same as, or greater than **s2**. Returns `<0` if **s1** is an initial substring of **s2.**

CPStrIndex

syntax `PStr` `CPStrIndex(s1h, index);`

`CPStrIndex` **returns** a pointer to the Pascal string denoted by **index** in a list of strings. If **index** is greater than or equal to the number of strings in the list, the function **returns** the pointer to the last string.

Parameter	**Type**	**Description**
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.

index	`int32`	Number of the string that you want, with 0 as the first string.

returns A pointer to the specified Pascal string.

CPStrInsert

`syntax` `MgErr` `CPStrInsert(s1h, s2, index);`

`CPStrInsert` inserts a new Pascal string before the **index** numbered Pascal string in a concatenated list of Pascal strings. If **index** is greater than or equal to the number of strings in the list, the function places the new string at the end of the list. `CPStrInsert` resizes the list to make room for the new string.

Parameter	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
s2	`PStr`	Pointer to a Pascal string.
index	`int32`	Position that you want the new Pascal string to have in the list of Pascal strings, with 0 as the first string.

returns `mFullErr` if there is not enough memory. Returns `noErr` otherwise.

CPStrLen *Macro*

syntax `int32` `CPStrLen(sp);`

`CPStrLen` **returns** the number of Pascal strings in a concatenated list of Pascal strings (that is, `sp->cnt`). Use the `CPStrSize` function to get the total number of characters in the list.

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

returns The number of strings in the concatenated list of Pascal strings.

CPStrRemove

syntax `void` `CPStrRemove(s1h, index);`

`CPStrRemove` removes a Pascal string from a list of Pascal strings. If **index** is greater than or equal to the number of strings in the list, the function removes the last string. `CPStrRemove` resizes the list after removing the string.

Parameter	Type	Description
s1h	`CPStrHandle`	Handle to a concatenated list of Pascal strings.
index	`int32`	Number of the string that you want to remove, with 0 as the first string.

CPStrReplace

syntax `MgErr` `CPStrReplace(s1h, s2, index);`

CPStrReplace replaces a Pascal string in a concatenated list of Pascal strings with a new Pascal string.

Parameter	Type	Description
s1h	CPStrHandle	Handle to a concatenated list of Pascal strings.
s2	PStr	Pointer to a Pascal string.
index	int32	Number of the string that you want to replace, with 0 as the first string.

returns mFullErr if there is not enough memory. Returns noErr otherwise.

CPStrSize

syntax `int32 CPStrSize(sp);`

CPStrSize **returns** the number of characters in a concatenated list of Pascal strings. Use the CPStrLen function to get the number of Pascal strings in the concatenated list.

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

returns The number of characters in the concatenated list of Pascal strings.

CToPStr

syntax `int32 CToPStr(cstr, pstr);`

CToPStr converts a C string to a Pascal string. This function works even if the pointers **cstr** and **pstr** refer to the same memory location. If the length of **cstr** is greater than 255 characters, the function converts only the first 255 characters. The function assumes that **pstr** is large enough to contain **cstr**.

Parameter	Type	Description
cstr	CStr	Pointer to a C string.
pstr	PStr	Pointer to a Pascal string.

returns The length of the string, truncated to a maximum of 255 characters.

FileNameCmp *Macro*

syntax `int32 FileNameCmp(s1, s2);`

FileNameCmp lexically compares two file names, to determine whether one is less than, the same as, or greater than the other. This comparison uses the same case sensitivity as the file system (that is, case insensitive for the Macintosh and the PC, case sensitive for the Sun SPARCstation).

Parameter	Type	Description
s1	PStr	Pointer to a Pascal string.
s2	PStr	Pointer to a Pascal string.

returns <0, 0, or >0 if **s1** is less than, the same as, or greater than **s2**. Returns <0 if **s1** is an initial substring of **s2**.

FileNameIndCmp *Macro*

syntax `int32 FileNameIndCmp(s1p, s2p);`

`FileNameIndCmp` is the same as `FileNameCmp`, except you pass the function handles to the string data instead of pointers. You can use `FileNameIndCmp` to compare two file names and lexically determine whether one is less than, the same as, or greater than the other. This comparison uses the same case sensitivity as the file system (that is, case insensitive for the Macintosh and the PC, and case sensitive for the Sun SPARCstation).

Parameter	Type	Description
s1p	`PStr *`	Pointer to a Pascal string.
s2p	`PStr *`	Pointer to a Pascal string.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2**.

FileNameNCmp *Macro*

syntax `int32 FileNameNCmp(s1, s2, n);`

`FileNameNCmp` lexically compares two file names to determine whether one is less than, the same as, or greater than the other, limiting the comparison to **n** characters. This comparison uses the same case sensitivity as the file system (that is, case insensitive for the Macintosh and the PC, case sensitive for the Sun SPARCstation).

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.
n	`uInt32`	Maximum number of characters to compare.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2**.

HexChar

syntax `int32 HexChar(n);`

`HexChar` **returns** the ASCII character in hex that represents the specified value **n** (0<=**n**<=15).

Parameter	Type	Description
n	`int32`	Decimal value between 0 and 15.

returns The corresponding ASCII hex character. If **n** is out of range, the ASCII character corresponding to **n** modulo 16 is returned.

IsAlpha

syntax `Bool32 IsAlpha(c);`

`IsAlpha` **returns** `TRUE` if the character **c** is a lowercase or uppercase letter (that is, in the set a to z or A to Z). On the SPARCstation, this function also **returns** `TRUE` for international characters (ˆ, ‡, €, and so on).

Parameter	Type	Description
c	`char`	Character that you want to analyze.

returns `TRUE` if the character is an alphabetic character, and `FALSE` otherwise.

IsDigit

syntax `Bool32` `IsDigit(c);`

`IsDigit` returns `TRUE` if the character **c** is between 0 and 9.

Parameter	Type	Description
c	`char`	Character that you want to analyze.

returns `TRUE` if the character is a numerical digit, and `FALSE` otherwise.

IsLower

syntax `Bool32` `IsLower(c);`

`IsLower` **returns** `TRUE` if the character **c** is a lowercase letter (that is, in the set a to z). On the SPARCstation, this function also **returns** `TRUE` for lowercase international characters (, š, and so on).

Parameter	Type	Description
c	`char`	Character that you want to analyze.

returns `TRUE` if the character is a lowercase letter, and `FALSE` otherwise.

IsUpper

syntax `Bool32` `IsUpper(c);`

`IsUpper` returns `TRUE` if the character **c** is between an uppercase letter (that is, in the set A to Z). On the SPARCstation, this function also **returns** `TRUE` for uppercase international characters (î, €, and so on).

Parameter	Type	Description
c	`char`	Character that you want to analyze.

returns TRUE if the character is an uppercase letter, and FALSE otherwise.

LStrBuf*Macro*

syntax `uChar` `*LStrBuf(s);`

`LStrBuf` returns the address of the string data of a long Pascal string (that is, the address of `s->str`).

Parameter	Type	Description
s	`LStrPtr`	Pointer to a long Pascal string.

returns The address of the string data of the long Pascal string.

LStrCmp

syntax `LStrPtr` `LStrCmp(l1p, l2p);`

`LStrCmp` lexically compares two long Pascal strings to determine whether one is less than, the same as, or greater than the other. This comparison is case sensitive.

Parameter	Type	Description
l1p	`LStrPtr`	Pointer to a long Pascal string.
l2p	`LStrPtr`	Pointer to a long Pascal string.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2.**

LStrLen *Macro*

syntax `int32` `LStrLen(s);`

`LStrLen` returns the length of a long Pascal string (that is, `s->cnt`).

Parameter	Type	Description
s	`LStrPtr`	Pointer to a long Pascal string.

returns The number of characters in the long Pascal string.

LToPStr

syntax `int32` `LToPStr(lstrp, pstr);`

`LToPStr` converts a long Pascal string to a Pascal string. If the long Pascal string is more than 255 characters, the function converts only the first 255 characters. This function works even if the pointers **lstrp** and **pstr** refer to the same memory location. The function assumes that **pstr** is large enough to contain **lstrp**.

Parameter	Type	Description
lstrp	`LStrPtr`	Pointer to a long Pascal string.
pstr	`PStr`	Pointer to a Pascal string.

returns The length of the string, truncated to a maximum of 255 characters.

PPStrCaseCmp

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

`PPStrCaseCmp` is the same as `PStrCaseCmp`, except you pass the function handles to the string data instead of pointers. You can use `PPStrCaseCmp` to compare two Pascal strings lexically and determine whether one is less than, the same as, or greater than the other. This comparison ignores differences in case.

Parameter	Type	Description
s1p	`PStr *`	Pointer to a Pascal string.
s2p	`PStr *`	Pointer to a Pascal string.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2.**

PPStrCmp

syntax `int32 PPStrCmp(s1p, s2p);`

`PPStrCmp` is the same as `PStrCmp`, except you pass the function handles to the string data instead of pointers. You can use `PPStrCmp` to compare two Pascal strings lexically and determine whether one is less than, the same as, or greater than the other. This comparison is case sensitive.

Parameter	Type	Description
s1p	`PStr *`	Pointer to a Pascal string.
s2p	`PStr *`	Pointer to a Pascal string.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2.**

PStrBuf *Macro*

syntax `uChar *PStrBuf(s);`

`PStrBuf` **returns** the address of the string data of a Pascal string (that is, the address following the length byte).

Parameter	Type	Description
s	`PStr`	Pointer to a Pascal string.

PStrCaseCmp

syntax `int32 PStrCaseCmp(s1, s2);`

`PStrCaseCmp` lexically compares two Pascal strings to determine whether one is less than, the same as, or greater than the other. This comparison ignores differences in case.

Parameter	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2.**

PStrCat

syntax `int32` `PStrCat(s1, s2);`

`PStrCat` concatenates a Pascal string, **s2**, to the end of another Pascal string, **s1**, and places the result in **s1**. This function assumes that **s1** is large enough to contain the resulting string. If the resulting string is larger than 255 characters, then `PStrCat` limits the resulting string to 255 characters.

Parameter	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

returns The length of the resulting string.

PStrCmp

syntax `int32` `PStrCmp(s1, s2);`

`PStrCmp` lexically compares two Pascal strings to determine whether one is less than, the same as, or greater than the other. This comparison is case sensitive.

Parameter	Type	Description
s1	`PStr`	Pointer to a Pascal string.
s2	`PStr`	Pointer to a Pascal string.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2.**

PStrCpy

syntax `PStr` `PStrCpy(dst, src);`

`PStrCpy` copies the Pascal string **src** to the Pascal string **dst**. This function assumes that the destination string is large enough to contain the source string.

Parameter	Type	Description
dst	`PStr`	Pointer to a Pascal string.
src	`PStr`	Pointer to a Pascal string.

returns A copy of the destination Pascal string pointer.

PStrLen*Macro*

syntax `uInt8 PStrLen(s);`

`PStrLen` **returns** the length of a Pascal string (that is, the value at the first byte at the specified address).

Parameter	Type	Description
s	`PStr`	Pointer to a Pascal string.

PStrNCpy

syntax `PStr PStrNCpy(dst, src, n);`

`PStrNCpy` copies the Pascal string **src** to the Pascal string **dst**. If the source string is greater than **n**, the function copies only **n** bytes. This function assumes that the destination string is large enough to contain the source string.

Parameter	Type	Description
dst	`PStr`	Pointer to a Pascal string.
src	`PStr`	Pointer to a Pascal string.
n	`int32`	Maximum number of bytes to copy including the length byte.

returns A copy of the destination Pascal string pointer.

PToCStr

syntax `int32 PToCStr(pstr, cstr);`

`PToCStr` converts a Pascal string to a C string. This function works even if the pointers **pstr** and **cstr** refer to the same memory location. This function assumes that **cstr** is large enough to contain **pstr**.

Parameter	Type	Description
pstr	`PStr`	Pointer to a Pascal string.
cstr	`CStr`	Pointer to a C string.

returns The length of the string.

PToLStr

syntax `int32 PToLStr(pstr, lstrp);`

`PToLStr` converts a Pascal string to a long Pascal string. This function works even if the pointers **pstr** and **lstrp** refer to the same memory location. The function assumes that **lstrp** is large enough to contain **pstr**.

Parameter	Type	Description
pstr	`PStr`	Pointer to a Pascal string.
lstrp	`LStrPtr`	Pointer to a long Pascal string.

returns The length of the string.

SPrintf
SPrintfp
PPrintf
PPrintfp
FPrintf
LStrPrintf

syntax

```
int32          SPrintf(CStr destCSt, CStr cfmt, ...);
int32          SPrintfp(CStr destCSt, PStr pfmt, ...);
int32          PPrintf(PStr destPSt, CStr cfmt, ...);
int32          PPrintfp(PStr destPSt, PStr pfmt, ...);
int32          FPrintf(File destFile, CStr cfmt, ...);
MgErr          LStrPrintf(LStrHandle destLsh, CStr cfmt,...);
```

All these functions format data into an ASCII format to a specified destination. A format string describes the desired conversions. These functions take a variable number of arguments, and each argument follows the format string paired with a conversion specification embedded in the format string. The second parameter, **cfmt** or **pfmt**, must be cast appropriately to either type `CStr` or `PStr`.

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

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

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

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

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

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

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

These functions accept the following standard formats and special characters.

- Special characters that can be embedded in strings:

`\b` backspace

`\f` form feed

`\n` new line (inserts the system-dependent end-of-line char(s); for example, CR on Macintosh, NL on UNIX, CRNL on DOS)

`\r` carriage return

`\s` space

`\t` tab

`%%` percentage character (to print `%`)

- Format arguments:

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

`[-]` Left-justifies what is printed; if not specified, the data is right-justified.

`[field size]` Specifies the minimum width of the field to print into. If not specified, this defaults to 0. If there is less than the specified number of characters in the data to print, the function pads with spaces on the left if you specified -; otherwise the function pads on the right.

`[.precision]` Sets the precision for floating-point numbers (that is, the number of characters after the decimal place). For strings, this specifies the maximum number of characters to print.

`[argument size]` Specifies the data size for an argument. It applies only to the **d**, **o**, **u**, and **x** conversion specifiers. By default, the conversion for one of the specifiers is from a word (16-bit integer). The flag **l** causes this conversion to convert the data so that the function assumes the data is a long integer value.

`[conversion]`

`b` binary

`c` print a character (%2c, %4c print on int16, int32 as a 2,4 char constant)

`d` decimal

`e` exponential

`f` fixed point format

`H` string handle (LStrHandle)

`o` octal

`p` Pascal string

`P` long Pascal string (LStrPtr)

`q` print a point (passed by value) as %d,%d representing horizontal, vertical coordinates

`Q` print a point (passed by value) as hv(%d,%d) representing horizontal, vertical coordinates

`r` print a rectangle (passed by reference) as %d,%d,%d,%d representing top,left, bottom, right coordinates

`R` print a rectangle (passed by reference) as tlbr(%d,%d,%d,%d) representing top,left, bottom, right coordinates

`s` string

`u` unsigned decimal

`x` hex

`z` Path

Any of the numeric conversion characters (`x, o, d, u, b, e, f`) can be preceded by {`cc`} to indicate that the number is passed by reference. `cc` can be `iB`, `iW`, É , `cX` depending on the corresponding numeric type. If `cc` is an asterisk (*) the numeric type (`iB` through `cX`) is an `int16` in the argument list.

StrCat

syntax `int32` `StrCat(s1, s2);`

`StrCat` concatenates a C string, **s2**, to the end of another C string, **s1**, placing the result in **s1**. This function assumes that **s1** is large enough to contain the resulting string.

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.

returns The length of the resulting string.

StrCmp

syntax `int32` `StrCmp(s1, s2);`

`StrCmp` lexically compares two strings to determine whether one is less than, the same as, or greater than the other.

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2**.

StrCpy

syntax `CStr` `StrCpy(dst, src);`

`StrCpy` copies the C string **src** to the C string **dst**. This function assumes that the destination string is large enough to contain the source string.

Parameter	Type	Description
dst	`CStr`	Pointer to a C string.
src	`CStr`	Pointer to a C string.

returns A copy of the destination C string pointer.

StrLen

syntax `int32` `StrLen(s);`

`StrLen` **returns** the length of a C string.

Parameter	Type	Description
s	`CStr`	Pointer to a C string.

returns The number of characters in the C string, not including the `NULL` terminating character.

StrNCaseCmp

syntax `int32` `StrNCaseCmp(s1, s2, n);`

`StrNCaseCmp` lexically compares two strings to determine whether one is less than, the same as, or greater than the other, limiting the comparison to **n** characters. `StrNCaseCmp` ignores differences in case in performing the comparison.

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.

n	`uInt32`	Maximum number of characters to compare.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2**.

StrNCmp

syntax `int32` `StrNCmp(s1, s2, n);`

`StrNCmp` lexically compares two strings to determine whether one is less than, the same as, or greater than the other, limiting the comparison to **n** characters.

Parameter	Type	Description
s1	`CStr`	Pointer to a C string.
s2	`CStr`	Pointer to a C string.
n	`uInt32`	Maximum number of characters to compare.

returns <`0`, `0`, or >`0` if **s1** is less than, the same as, or greater than **s2**. Returns <`0` if **s1** is an initial substring of **s2**.

StrNCpy

syntax `CStr` `StrNCpy(dst, src, n);`

`StrNCpy` copies the C string **src** to the C string **dst**. If the source string is less than **n** characters, the function pads the destination with null characters. If the source string is greater than **n**, then only **n** characters are copied. This function assumes that the destination string is large enough to contain the source string.

Parameter	Type	Description
dst	`CStr`	Pointer to a C string.
src	`CStr`	Pointer to a C string.
n	`int32`	Maximum number of characters to copy.

returns A copy of the destination C string pointer.

ToLower

syntax `uChar` `ToLower(c);`

`ToLower` **returns** the lowercase value of **c** if **c** is an uppercase alphabetic character. Otherwise, it **returns c** unmodified. On the SPARCstation, this function also works for international characters (€ -> Š, and so on).

Parameter	Type	Description
c	`uChar`	Character that you want to analyze.

returns The lowercase value of **c**.

ToUpper

syntax `uChar ToUpper(c);`

`ToUpper` **returns** the uppercase value of **c** if **c** is a lowercase alphabetic character. Otherwise, it **returns** **c** unmodified. On the SPARCstation, this function also works for international characters (Š -> €, and so on).

Parameter	**Type**	**Description**
c	`uChar`	Character that you want to analyze.

returns The uppercase value of **c**

Utility Functions

Click here to view a list of all Support Manager Functions.

BinSearch

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

`BinSearch` searches an array of an arbitrary data type using the binary search algorithm. In addition to passing the array that you want to search to this routine, you also pass a comparison procedure that this sort routine then uses to compare elements in the array.

The comparison routine should return a number less than zero if **a** is less than **b**, zero if **a** is equal to **b**, and a number greater than zero if **a** is greater than **b**.

You should declare the comparison routine to have the following parameters and return type.

int32 compareProcP(UPtr a, UPtr b);

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

returns The position in the array where the data is found (with `0` being the first element of the array), if it is found. If the data is not found, BinSearch returns `-i-1`, where `i` is the position where `x` should be placed.

QSort

syntax `void` `QSort(arrayp,n,elmtSize, compareProcP());`

`QSort` sorts an array of an arbitrary data type using the QuickSort algorithm. In addition to passing the array that you want to sort to this routine, you also pass a comparison procedure that this sort routine then uses to compare elements in the array.

The comparison routine should return a number less than zero if **a** is less than **b**, zero if **a** is equal to **b**, and a number greater than zero if **a** is greater than **b**.

You should declare the comparison routine to have the following parameters and return type.

int32 compareProcP(UPtr a, UPtr b);

Parameter	Type	Description
arrayp	`UPtr`	Pointer to an array of data.
n	`int32`	Number of elements in the array that you want to sort.
elmtSize	`int32`	Size in bytes of an array element.
compareProcP	`procPtr`	Comparison routine that you want `QSort` to use to compare array elements. `QSort` passes this routine the addresses of two elements that it needs to compare.

Unused *Macro*

syntax `void` `Unused(x)`

`Unused` indicates that a function parameter or local variable is not used by that function. This is useful for suppressing compiler warnings for many compilers. Notice that no semicolon is used with this macro.

Parameter	Type	Description
x	-	Unused parameter or local variable.

Time Functions

Click here to view a list of all Support Manager Functions.

ASCIITime

syntax `CStr ASCIITime(secs);`

`ASCIITime` returns a pointer to a string representing the date and time of day corresponding to *t* seconds after January 1, 1904, 12:00 AM, GMT. This function uses the same date format as that returned by the `DateCString` function using a mode of 2. The date is followed by a space, and the time is in the same format as that returned by the `TimeCString` function using a mode of 0. As an example, this function might return Tuesday, Dec 22, 1992 5:30. On the SPARCstation, this function accounts for international conventions for representing dates.

Parameter	**Type**	**Description**
secs	uInt32	Seconds since the January 1, 1904, 12:00 AM, GMT.

returns The date and time as a C string.

DateCString

syntax `CStr DateCString(secs, fmt);`

Note: This function was formerly called `DateString`**.**

`DateCString` **returns** a pointer to a string representing the date corresponding to *t* seconds after January 1, 1904, 12:00 AM, GMT. On the SPARCstation, this function accounts for international conventions for representing dates.

Parameter	**Type**	**Description**
secs	`uInt32`	Seconds since January 1, 1904, 12:00 AM, GMT.
fmt	`int32`	Code describing the format for the returned string. This parameter determines the format of the returned date string and can have the following values.

Fmt	**Meaning**
0	Return the date in short date format, *mm*/*dd*/*yy*, where *mm* is a number between 1 and 12 representing the current month, *dd* is the current day of the month (1 through 31), and *yy* is the last two digits of the

corresponding year. An example is 12/31/92.

1 Return the date in long date format, *dayName*, *MonthName*, *DayOfMonth*, *LongYear*. An example is Thursday, December 31, 1992.

2 Return the date in abbreviated date format, *AbbrevDayName*, *AbbrevMonthName*, *DayOfMonth*, *LongYear*. An example is Thu, Dec 31, 1992.

returns The date as a C string.

DateToSecs

syntax `uInt32` `DateToSecs(dateRecordP);`

`DateToSecs` converts from a time described using the DateRec data structure to the number of seconds since January 1, 1904, 12:00 AM, GMT.

Parameter	**Type**	**Description**
dateRecordP	`DateRec *`	Pointer to a `DateRec` structure. `DateToSecs` stores the converted date in the fields of the date structure referred to by **dateRecordP**. See the *Pointers as Parameters* section of Chapter 1, *CIN Overview*, in the *Code Interface Reference Manual* for more information about using this parameter.

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

MilliSecs

syntax `uInt32` `MilliSecs();`

returns The time since an undefined system time in milliseconds. The actual resolution of this timer is system dependent.

SecsToDate

syntax `void` `SecsToDate(secs, dateRecordP);`

`SecsToDate` converts the seconds since January 1, 1904, 12:00 AM, GMT into a data structure containing numerical information about the date, including the year (1904 through 2040), the month (1 through 12), the day as it corresponds to the current year (1 through 366), month (1 through 31), and week (1 through 31), hour (0 through 23), the hour (0 through 23), minute (0 through 59), and second (0 through 59) of that day, and a value indicating whether the time specified uses daylight savings time.

Parameter	**Type**	**Description**
secs	`uInt32`	Seconds since January 1, 1904, 12:00 AM, GMT.
dateRecordP	`DateRec *`	Pointer to a `DateRec` structure. `SecsToDate` stores the converted date in the fields of the date structure referred to by **dateRecordP**. See the *Pointers as Parameters* section of Chapter 1,

CIN Overview, in the *Code Interface Reference Manual* for more information about using this parameter.

TimeCString

syntax `CStr` `TimeCString(secs, fmt);`

Note: This function was formerly called `TimeString`.

`TimeCString` **returns** a pointer to a string representing the time of day corresponding to *t* seconds after January 1, 1904, 12:00 AM, GMT. On the SPARCstation, this function accounts for international conventions for representing dates.

Parameter	Type	Description
secs	`uInt32`	Seconds since January 1, 1904, 12:00 AM, GMT.
fmt	`int32`	Code describing the format for the returned string. The parameter **fmt** determines the format of the returned time string and can have the following values.

Fmt	Meaning
0	Return the time in the format *hh*:*mm*. The first value, *hh*, represents the hour (0 through 23, with 0 as midnight), and the second value, *mm*, represents the minute (0 through 59).
1	Return the time in the format *hh*:*mm*:*ss*. The first value, *hh*, represents the hour, the second value, *mm*, represents the minute (0 through 59), and the third value, *ss*, represents the second (0 through 59).

returns The time as a C string.

TimeInSecs

syntax `uInt32` `TimeInSecs();`

returns The current date and time in seconds relative to January 1, 1904, 12:00M AM, Greenwich mean time (GMT).

CIN Function Overview

This topic includes an overview of CIN functions. For specific function information, see the following topics:

Memory Manager Functions
File Manager Functions
Support Manager Functions

Included with Code Interface Nodes (CINs) are a large set of external functions you can use to perform simple and complex operations. These functions organized into libraries called managers, range from low-level byte manipulation to routines for sorting data and managing memory. All CIN manager routines are platform-independent. If you use these routines, you can create CINs that will work on all platforms that LabVIEW supports.

The CIN managers include routines for memory, file, and support.

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

The file manager routines include operations for creating, opening, and closing files, writing data to files, and reading data from files.   In addition, file manager routines allow you to create directories, determine characteristics of files and directories, and copy files.

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

For more general information on all the manager routines, refer to Chapter 5, *Manager Overview*, of the *Code Interface Reference Manual*.